Towards Extremely Fast Context Switching in a Block-Multithreaded Processor

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

Multithreaded processors use a fast context switch to bridge latencies caused by memory accesses or by synchronization operations. In the block-multithreaded processor – called Rhamma – 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 destined for another unit. Switching contexts on each load/store instruction sequence allows a much faster context switch in the execution unit than previously published designs do. The results show the potential of multithreading to spare expensive off-chip cache in a workstation environment. The load/store unit proves as the principal bottleneck. In particular the memory cycle time is performance critical. We show that multithreaded processors profit more than conventional RISC processors by a shorter memory cycle time.

1. Introduction

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. Since processor cycle times shrink faster than RAM access times, standard microprocessors suffer from data shortage unless provisions are made to bridge latencies caused by memory accesses or by synchronization operations. On-chip and off-chip caches in combination with prefetching techniques lessen the gap of the processor-memory access, but cannot completely remove the data shortage.

Our research project aims at the development of a processor usable in a single- or a multi-processor workstation, running in a multithreaded operating system environment. The access of data and the synchronization of threads cause processor idle times. It is the object of our research to fill these idle times by extremely fast switches between different threads of control. We further implement suitable synchronization primitives that prevent busy waiting.

Related approaches are the HEP [1], Horizon [2] and Tera systems [3], the Sparcle processor of the MIT Alewife machine [4], the *T [5], and data flow architectures [6] in general. HEP, Horizon and Tera are finely grained multithreaded processors that switch context on every instruction paying with a bad single thread performance. The Sparcle processor switches context on a cache miss that causes a remote memory access. Data cache misses are detected in a late stage of the processor pipeline, thus several pipeline stages have to be reloaded, causing a context switch overhead of 14 cycles. The *T does not incorporate multiple register sets, therefore a context switch causes an overhead of load and store operations. Principally, all these approaches differ from our approach by assuming a multiprocessor environment with a much higher memory access time. The processor should be able to bridge memory latencies and synchronization waiting times so efficiently that it is appropriate for use in a single–processor workstation.

The multithreaded processor of the Media Research Laboratory of Matsushita Electric Industrial Co. [7] uses the simultaneous multithreading [8] technique. Instructions of several threads are simultaneously issued to the functional units of a superscalar processor. This approach shows the best performance gains, but also high overhead for the dispatch unit.

Another related proposition is the decoupled architecture, which achieves high scalar perfomance by cleanly splitting instruction processing into memory access and execution tasks [9]. This architecture has two separate sets of instructions executed by different units. The units communicate via “architectural queues”.

Our approach – called Rhamma – is a combination of the block-multithreading technique and the decoupled architecture approach. It is similar to the Sparcle processor. 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 switch 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 from 1 to 5 processor cycles are presented and their hardware costs discussed.

The next section describes the Rhamma processor and its behavior in detail, Section 3 the simulator and workload, and Section 4 the simulation results concerning the application in a single-processor workstation.

2. The Rhamma Processor

2.1 Overview

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 are predictable. They depend on the memory hierarchy of the microcomputer or workstation. The latter depend on the program execution and are nonpredictable. 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. The load/store unit detects the end of a memory access by load/store acknowledgements. The units are coupled by FIFO buffers and access different register sets. Figure 1 shows the microarchitecture of the multithreaded 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. Activation frames of blocked threads are stored in the memory if more activation frames exist than register sets are available.

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.

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. Access to the new thread tag and loading the new instruction pointer from the thread tag, respectively, need one cycle each. The first instruction of the new thread is decoded after two more cycles - thus context switching overhead sums up to 5 cycles.

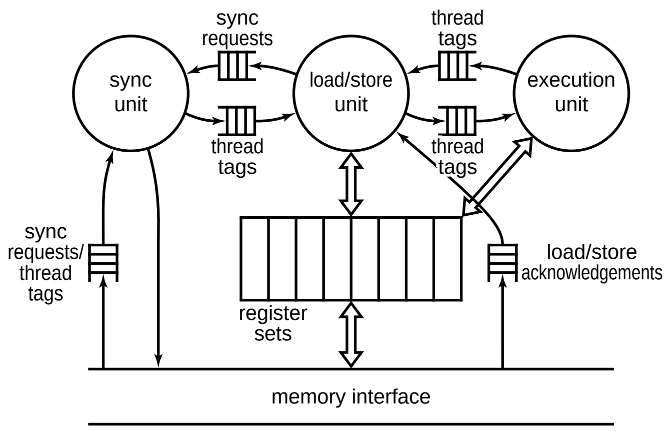

Figure 1: Microarchitecture of the Rhamma processor

However, context switching overhead can be reduced to one clock cycle: Coding the context switch in the preceding instruction saves the waste of the first cycle (decode an instruction that can not be executed by the unit). If enough threads are loaded on the processor, access to the next thread tag and loading the respective instruction pointer in advance saves two more cycles. Duplication of the instruction fetch stage and preloading the next instruction reduces context switching overhead to a single cycle. If the decode stage is also doubled, context switching overhead would even be zero.

The main bottleneck of each high-performance processor is the unit which executes load and store instructions. In a multithreaded processor the load/store bottleneck is even more essential than in a conventional processor due to the higher throughput of data. But multithreading allows new possibilities to solve the load/store bottleneck:

Several load/store requests are send to the memory (provided that the load/store instructions are data-independent of each other and that the cycle time is less than the access time). Then the thread tag is handed over to the execution unit, sync unit respectively. The next execution instructions are executed if the instructions are data-independent from the previous ones. Depending on scoreboarding bits in the register set the execution unit or the load/store unit stall on a data-dependent instruction. In the case the load/store unit or the execution unit has to stall for a dependent instruction, we use a finely grained multithreading technique: the unit switches the thread of control, and the instructions of the independent thread are scheduled. Execution of the succeeding instructions of the switched thread is resumed by the execution unit depending on the scoreboarding bit, respectively by the load/store unit after receiving the acknowledgement corresponding to the memory request.

2.2 Behavioral Description

□The instruction set architecture of the Rhamma processor is derived from the DLX instruction set [10] extended by instructions for synchronization purpose (lock and unlock), by move instructions (mvr, mvl) and by thread management instructions (start, yield, stop, getfr and relfr). This subsection gives a detailed microarchitecture description of the multithreaded Rhamma processor using a description language syntactically similar to Modula–2.

□The load/store unit (see figure 2) retrieves a thread tag F from the FIFO buffer, loads the program counter, fetches the designated instruction, and decodes it. The address of the instruction is located in the register PCReg of the activation frame F. Register sets are modelled by a twodimensional shared array R. The first index maps the thread tag on its activation frame. The second index selects the register in the activation frame. Each instruction is represented by a tuple (op, r1, r2, r3, imm) with operation op, registers r1, r2, r3 and the constant value imm. The instruction dependencies are determined by testing scoreboarding bits R[F, SbReg] in case of a pending load and a counter R [F, CtReg] in case of pending store instructions. If the test fails, the thread tag F is stored again in one of the FIFO buffers that are read by the load/store unit (hand F over to load/store). Thereby the thread is rescheduled by the load/store unit later.

□For each memory operation the requests loadRequ, storeRequ and exchRequ are sent to the memory interface (storeRequ and exchRequ transport data to be stored). Memory then returns a loadAckn, storeAckn or exchAckn acknowledgement (loadAckn and exchAckn transport data to be loaded). Load, store and exchange requests are sent by the load/store unit until the next instruction is an execution or synchronization instruction. Then the load/store unit switches the actual thread (EXIT inner loop). The old thread is resumed when the load/store unit receives the corresponding acknowledgement. Therefore the thread tag is necessary as an argument of each request.

□The unlock instruction does not trigger a context switch.

```
SHARED    R  : ARRAY THREADTAG, REGISTER OF WORD;

UNIT load/store;

    UNIT req;
    BEGIN
        LOOP
            LOOP
                pc := R[ F, PCReg ];
                (op, r1, r2, r3, imm) := load instruction at pc;
                pc := next pc;
                CASE op OF
                |   add..getfr:   hand F over to execution;
                                  EXIT inner loop;
                |   lw:         IF (r1, r2 IN R[ F,SbReg ]) OR (R[ F,CtReg ] ≠ 0) THEN
                                      hand F over to load/store;
                                      EXIT inner loop;
                                  ELSE
                                      INCL( R[ F, SbReg ], r2);
                                      hand (F, loadRequ, R[F, r1]+imm, r2)
                                                                       over to memory;
                                  END;
                |   sw:           IF (r1, r2 IN R[ F, SbReg ]) OR (R[ F,CtReg] ≠ 0) THEN
                                      hand F over to load/store;
                                      EXIT inner loop;
                                  ELSE
                                      INC ( R[ F, CtReg ] );
                                      hand(F,storeRequ,R[F,r2]+imm, R[F,r1])
                                                                       over to memory;
                                  END;
                |   exch:         IF (r1, r2 IN R[ F, SbReg]) OR (R[ F,CtReg ] ≠ 0) THEN
                                      hand F over to load/store;
                                      EXIT inner loop;
                                  ELSE
                                      INCL( R[ F, SbReg ], r2); INC ( R[ F, CtReg ] );
                                      hand(F,exchRequ,R[F,r2]+imm,R[F,r1],r2)
                                                                       over to memory;
                                  END;
                |   lock:         IF (r1 IN R[ F, SbReg]) AND (R[ F, CtReg ] ≠ 0) THEN
                                      hand F over to load/store;
                                      EXIT inner loop;
                                  ELSE
                                      hand (F, lockRequ, R[F, r1]+imm) over to sync;
                                      R[ F, PCReg ] := pc;
                                      EXIT inner loop;
                                  END;
                |   unlock:       IF (R[ F, SbReg ] ≠ {}) OR (R[ F, CtReg ] ≠ 0) THEN
                                      hand F over to load/store;
                                      EXIT inner loop;
                                  ELSE
                                      hand (F, unlockRequ, R[F, r1]+imm) over to sync;
                                  END;
                END;
                R[ F, PCReg ] := pc;
    END END END req;

    UNIT ack;
    BEGIN
        LOOP
            take (F, code, a, data, reg) from memory;
            CASE code OF
            |   loadAckn:     R[ F, reg ] := data; EXCL ( R[ F, SbReg ], reg);
            |   storeAckn:    DEC ( R[ F, CtReg ] );
            |   exchAckn:     EXCL ( R[ F, SbReg ], reg); DEC ( R[ F, CtReg ] );
        END END;
    END ack;

END load/store;
```

Figure 2: The Load/Store Unit

The following instructions are executed at once avoiding blocking the actual thread. With all requests being confirmed (R[F, SbReg] = {}) AND (R[F, CtReg] = 0) the thread tag of the next instruction is handed over to the execution unit in case of an execution instruction, and the synchronization instruction is handed over to the sync unit otherwise. The thread is switched afterwards.

The execution unit (see figure 3) retrieves a thread tag F from the load/store buffer, loads the program counter, fetches the designated instruction, and decodes it.

If the instruction is an arithmetic or logic operation (add, sub, and, … , srli) and the instruction is independent of pending load/store instructions, the result is computed and written back to the destination register. In case the instruction has to access unwritten registers, the unit stops the execution of the thread and hands the thread tag over to the FIFO buffer to the execution unit (hand F over to execution). The instructions mvr and mvl are used to exchange the contents of registers of different activation frames. Branches are realized by the instruction beqz.

The instruction start activates the thread which is specified

```
SHARED R        : ARRAY THREADTAG, REGISTER OF WORD;
LOCAL   Free     : SET OF THREADTAG;

UNIT execution;
BEGIN
    LOOP
        take F from load/store;
        LOOP
            pc := R[ F, PCReg ];
            (op, r1, r2, r3, imm) := load instruction at pc;
            pc := next pc;
            CASE op OF
            |   add..srl:     IF r1, r2, r3 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  R[ F, r3 ] := R[ F, r1 ] op R[ F, r2 ];
                              END;
            |   addi..srli:   IF r1, r2 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  R[ F, r2 ] := R[ F, r1 ] op imm;;
                              END;
            |   mvr:          IF (r1,r3 IN R[ F,SbReg]) OR
                                              (r2 IN R[ R[F,r3 ], SbReg ]) THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  R[ R[ F, r3 ], r2 ] := R[ F, r1 ];
                              END;
            |   mvl:          IF (r3,r2 IN R[ F,SbReg]) OR
                                              (r1 N R[ R[ F, r2 ], SbReg ]) THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  R[ F, r3 ] := R[ R[ F, r2 ], r1 ];
                              END;
            |   beqz:         IF r1, r2 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSIF R[ F, r1 ] = Null THEN
                                  pc := R[ F, r2 ] + imm;
                              END;
            |   start:        IF r1 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  hand R[ F, r1 ] over to load/store;
                              END;
            |   yield:        IF r1 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  R[ F, PCReg ]:=pc; F := R[ F, r1 ];
                                  pc:=R[ F, PCReg ];
                              END;
            |   stop:         R[ F, PCReg ] := pc;
                              EXIT inner loop;
            |   relfr:        IF r1 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  add R[ F, r1 ] to Free;
                                  IF F in Free THEN EXIT inner loop; END
                              END;
            |   getfr:        IF r1, r2 IN R[ F, SbReg] THEN
                                  hand F over to execution;
                                  EXIT inner loop;
                              ELSE
                                  pick one G out of Free;
                                  R[F,r1] := G; R[G FrReg ] := G;
                                  R[G,SbReg]:={};R[G,CtReg]:=0;
                                  R[G,PCReg]:= R[F,r2] + imm;;
                              END;
            |   lw..unlock:   hand F over to load/store;
                              EXIT inner loop;
            END;
            R[ F, PCReg ] := pc;
END END END execution;
```

Figure 3: The Execution Unit

by the arguments of the start instruction by handing the thread tag over to the load/store unit. The instruction yield switches the running thread by the thread that is determined by the instruction. Activation frames are created and initialized by the instruction getfr, and released by relfr. The instruction stop suspends the actual thread. The execution succeeds with the instructions of the next thread tag.

The instructions lw, sw, exch, lock and unlock are executed by the load/store unit and by the sync unit, respectively. Thus the execution unit stops the actual thread similarly to the stop instruction and hands the thread tag over to load/store unit via the FIFO buffer.

The sync unit (see figure 4) supports the synchronization primitives lock and unlock. Each mutex variable exists only once. In case of a multiprocessor system each mutex variable is bound to the sync unit of a specific processor. Synchronization requests (lockRequ or unlockRequ) on mutex variables are sent to the corresponding sync unit.

```
LOCAL   mutex:    ARRAY WORD OF ( locked, unlocked );
        waiting:  ARRAY WORD OF SET OF THREADTAG;

UNIT sync;
BEGIN
   LOOP
      take (F, code, var) from  load/store or external sync;
      CASE code OF
      | lockRequ:     IF var at this sync THEN
                         IF mutex[var] = locked THEN
                            add F to waiting[var];
                         ELSE
                            mutex[var] := locked;
                            IF (F, code, var) from load/store THEN
                               hand F over to load/store;
                            ELSE
                               hand (F, lockAckn, var) over to external sync;
                         END END
                      ELSE
                         hand (F, code, var) over to external sync;
                      END;
      | unlockRequ:   IF var at this sync THEN
                         IF mutex[var] = locked THEN
                            IF waiting[var] = {} THEN
                               mutex[var] := unlocked;
                            ELSE
                               pick F out of waiting[var];
                               IF (F, code, var) from load/store THEN
                                  hand F over to load/store;
                               ELSE
                                  hand (F, lockAckn, var) over to external sync;
                         END END END
                      ELSE
                         hand (F, code, var) over to external sync;
                      END;
      |   lockAckn:hand F over to load/store;
END END END sync;
```

Figure 4: The Sync Unit

In case of a lock request the serving sync unit tests the state of the mutex variable. If the variable is in state "locked", the thread tag F is added to the set of threads waiting for the mutex variable. If the variable is in state "unlocked", the mutex variable is locked and the thread tag is returned by a lockAckn. Then the next synchronization request is performed.

In case of an unlock request, a thread tag is picked from the set of threads waiting for the lock variable, and the execution of the thread is resumed. If no waiting thread tag is left, the variable is unlocked.

3. The Simulator

We evaluate the multithreaded Rhamma processor versus a conventional processor without multithreading represented by the original DLX processor [10] using an event-driven simulation at the register-transfer level. Both, the multithreaded and the conventional processor overlap the execution of load/store instructions with the execution of independent execution instructions.

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 on-chip cache(s), off-chip cache, or memory.

Depending upon the memory hierarchy we distinguish access times, cycle times and hit rates of the on-chip cache, the off-chip cache and the main memory.

As simulation work load we used several small applications 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 work load was 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 was two. These simulation work load did not contain synchronization instructions.

Access and cycle times were chosen due to a workstation memory configuration in [11] using a 100 MHz PowerPC 604 [12]. Wang et al. [11] assume 9 ns synchronous burst

	access time	cycle time
on-chip cache	2	1
off-chip cache	9	9
DRAM memory	27	14-27

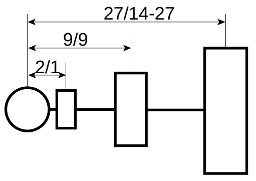

Table 1: Access and cycle times

SRAM with 3-1-1-1 burst read for the off-chip cache, and 60 ns DRAM with a 8-3-3-3 burst read for memory. The 66 MHz processor/memory bus allows split transactions and address pipelining. Therefore the memory cycle time was chosen as fraction of the memory access time. Access and cycle times are shown in table 1.

4. Simulation Results

The four diagrams in figure 5 compare performances of the conventional processor and of the Rhamma processor assuming context switch times of one, three and five cycles. The on-chip cache hit rate is fixed at 80%. We vary the combined hit rate of the on-chip and off-chip caches (assuming an inclusion principle) and the memory cycle time. The vertical axis shows the yielded simulation time steps for the executed benchmark program.

We see that memory access rate and context switch time are critical for the performance. The influence of the memory access rate is shown by the slope of the planes in figure 5 from left to right. A low memory access rate corresponds to a high combined cache hit rate (at the right side of the diagrams). The conventional processor profits more than the Rhamma processor from high cache hit rates. The realistic (combined) cache hit rates range from 95% to 99%. In this region the Rhamma processor with context switch time of one performs best and much better than the conventional processor. In figure 6 we compare

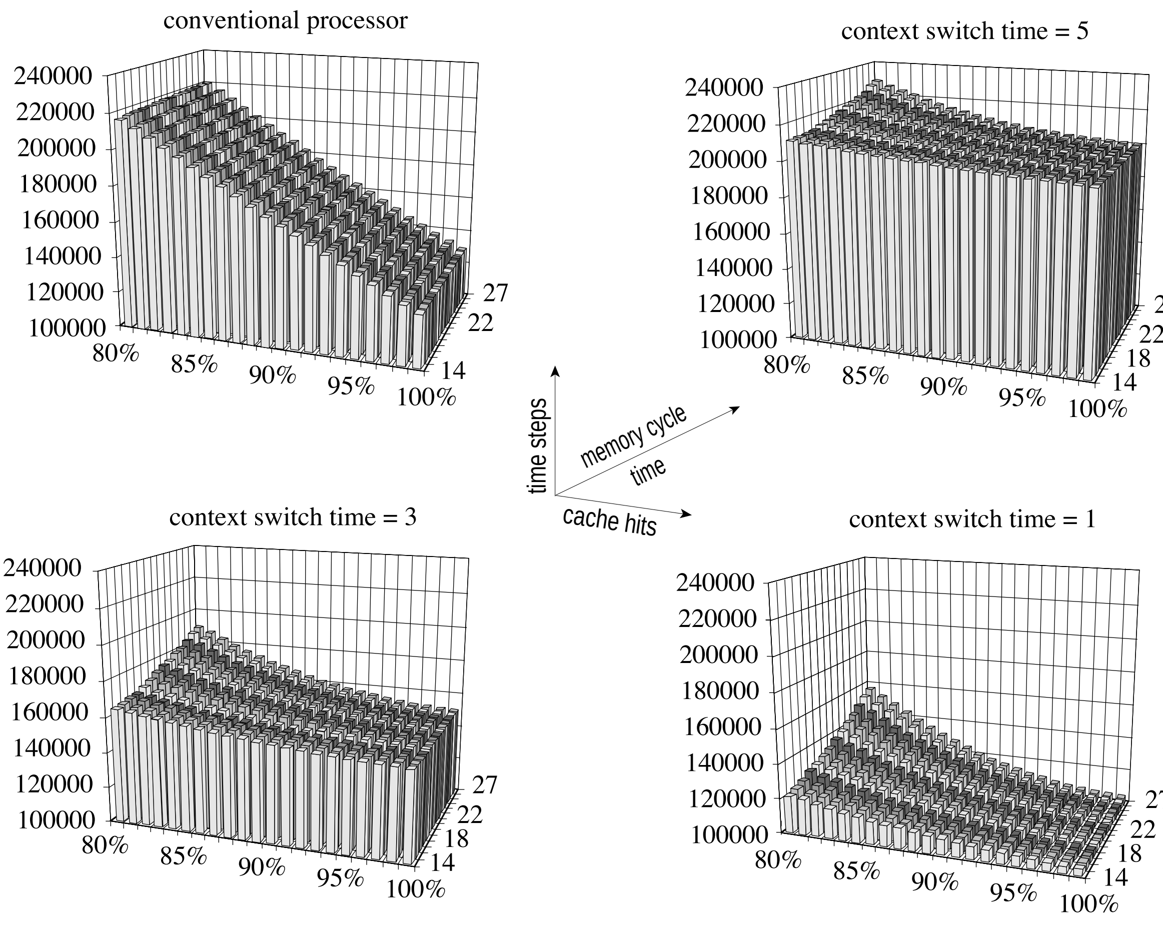

Figure 5: Conventional processor compared with Rhamma assuming context switch times of 1, 3 and 5 cycles.

the performances of the Rhamma processor without off-chip cache (varying context switch times from 1 to 5) with the conventional processor configuration assuming cache hit rates of 80% (no on-chip cache), 90% (combined on-chip off-chip caches), and 100% (all on-chip cache misses are served by the off-chip cache).

The conventional processor can only use a small part of the access time by cycle time ratio. A context switch time of five for the Rhamma processor is not enough to outperform the conventional processor. We see that a Rhamma processor with a context switch time of one or two processor cycles and an on-chip cache with 80% hit rate performs better than a conventional processor with the same on-chip cache and an ideal off-chip cache, i.e., multithreading saves the necessity of an expensive off–chip cache. Of course, using an off–chip cache would also increase the performance of the multithreaded 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 work load usable for access time bridging which is advantageous when long latencies (in case of synchronization operations, page misses, and reload via the PCI bus) have to be bridged by the multithreaded processor. For short latencies as assumed in the simulations above three threads proved to be sufficient.
- 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 access time bridging capability of the multithreaded processor.
- The memory cycle time proves as the critical parameter for the multithreaded processor. Increasing the cycle time slows down the performance as soon as the access time cannot completely be bridged by the multithreaded processor. A shorter cycle time widens the load/store bottleneck, thus possibly increasing performance.
- Increasing the context switch time worsens the performance of the multithreaded processor. However, repeating the simulations with the context switch time of more than five yields similar results if access times are increased, too.
- The measured performances depend on the number of data independent instructions following a load/store instruction. If enough threads are provided, the waiting time in the FIFO buffer to the execution unit is sufficient to bridge the access time. In contrast, the 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:

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

- Our multithreaded processor as well as the conventional processor is based on a scalar RISC processor. It is not easy to compare the simulation results with a hypothetical superscalar processor. 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 faster cycle times.

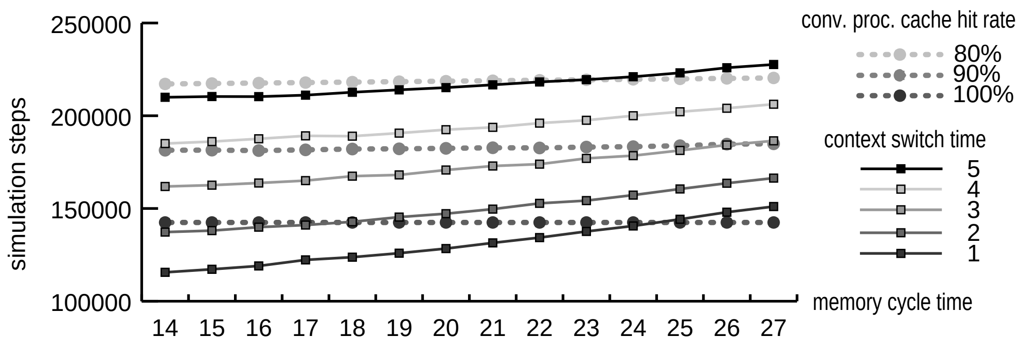

Figure 6: Conventional processor with off-chip cache compared with Rhamma without off-chip cache

5. Conclusions

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 switch 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. Applying the common access times, cycle times and hit ratios of a single processor workstation or personal computer, we show that expensive off-chip caches can be saved using a multithreaded processor. Because of the short context switch 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.

Processor design is a trade-off between performance gains and hardware costs. Our simulations give a performance estimation. To assess the hardware costs for reducing the context switch time, we work toward the hardware synthesis of different implementation alternatives.

References

[1] B. J. Smith: The Architecture of HEP. In: J. S. Kowalik (Ed.): Parallel MIMD Computation: The HEP Supercomputer 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] R. S. Nikhil, G. M. Papadopoulos, Arvind: *T: A Multithreaded Massively Parallel Architecture. 19th International Symposium on Computer Architecture, 1992, 156–167.

[6] Arvind, L. Bic, T. Ungerer: Evolution of Dataflow Computers. In: J.-L. Gaudiot, L. Bic (Eds.): Advanced Topics in Data-Flow Computing. Prentice-Hall 1991, 3 - 33.

[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] 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.

[9] J. E. 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.

[10] J. L. Hennessy, D. A. Patterson: Computer Architecture a Quantitative Approach, San Mateo 1996.

[11] K. Wang, et al.: Designing the MPC105 PCI Bridge/Memory Controller. IEEE Micro, April 1995, 44 - 49.

[12] B. Ryan, T. Thompson: PowerPC 604 Weighs In. BYTE, June 1994, 265 - 266.