Hardware Support for Fine-Grain Multi-Threading in LEON3

M. Daněk, L. Kafka, L. Kohout, J. Sýkora
ÚTIA AV ČR, v.v.i., Signal Processing, Pod Vodárenskou věží 4, Praha 8, 182 08, Czech Republic

***Abstract*— The article describes instruction set extensions for a variant of multi-threading called micro-threading for the LEON3 SPARCv8 processor. An architecture of the developed processor ispresented and its key blocks described - cache controller, register file, thread scheduler. The processor has been implemented in a Xilinx Virtex2Pro and Virtex5 FPGAs. The extensions are evaluated in terms of extra resources needed, and the overall performance of the developed processor is shown for a simple DSP computation typical for embedded systems.**

I. Introduction

Current processors have reached their maximum operating frequency, and performance improvements must be sought in better organization of the computation. One area for improvements is the tolerance of latency of data caused e.g. by a memory or I/O access, which is usually handled by context switching and executing computation threads that have data available in processors that support multi-threading.

As the silicon area becomes cheaper as a consequence of the Moore's law, it has become viable to extend processors to support in hardware execution of multiple threads on one processor or in a multiprocessor cluster. Two significant examples are the SUN Microsystems OpenSPARC T1/T2 and the MIPS MT processors. OpenSPARC T1/T2 is an open-source version of the UltraSPARC T1/T2 [1], [2]; T1 has been ported to the Xilinx FPGAs, while MIPS MT [3] is a commercial processor available as an ASIC. The architecture complexity of the open-source OpenSPARC T1/T2 is too high for embedded applications, which is due to their primary domain in server and desktop computing. Also the context switch time for T1/T2 is high, about 1000 clock cycles. We do not know of any other multithreaded processor available in source code to the design community.

To overcome this we have designed and implemented instruction set extensions for the simpler LEON3 SPARCv8 processor suitable for embedded applications. This paper describes the architecture of the modified LEON3 [4] processor (which we call UTLEON3) and the impact of the architectural improvements on the processor performance.

The main reasons for this work were to provide a feedback to the microthreaded model of computation and on current technology limitations so that the SW models and assumptions could be modified on the theoretical model. We were also interested in effects of the uT extensions on area and frequency, the input assumption being small area overhead and almost no impact on the clock frequency. We also wanted to compare the micro-threaded architecture with similar developments (OpenSPARC, MIPS MT) As GRLIB inherently compatible with ASIC tool flow, the path to ASIC implementation is open.

The paper is structured as follows: Section II describes the extra machine instructions that implement micro-threading in SPARC. Section III describes the architecture of the key blocks that implement the micro-threaded extensions. Section IV compares FPGA implementations of the classical LEON3 and the new UTLEON3 in terms of resource requirements. Section V evaluates the speedup of a simple assembly program in a legacy version executed on the classical LEON3 as well as the UTLEON3 processor, and a micro-threaded version executed on the UTLEON3 processor. Section VI compares UTLEON3 with the OpenSPARC T1/T2 processor. Section VII concludes the paper.

II. Micro-Threading

Micro-threading is a multi-threading variant that decreases the complexity of context management. The goal of micro-threading is to tolerate long-latency operations (LD/ST and multi-cycle operations such as floating-point) and to synchronize computation on register access. For an overview of multi-threading see [5].

In a simple case the context can be represented by the program counter and by window pointers to the register file. Micro-threading has been developed both on the assembly and C levels. The basic conceptual unit is a family of threads that share data and implement one piece of a computation. In a simple view one family corresponds to one for-loop in the classical C; in micro-threading each iteration (each thread) of a hypothetical for-loop (represented by a family of threads) is executed independently according to data dependencies. A family is synchronized on termination of all its threads. For more details on micro-threading see [6], [7], [8].

A possible speedup generated by micro-threading comes from the assumption that while one thread is waiting for its input data, another thread has its input data ready and can be scheduled in a few clock cycles and executed. Another assumption is that load and store operations themselves need not be blocking since the real problem arises just when an operation accesses a register that does not contain a valid data value. Finally, the thread management logic is considered simple enough to fit in the processor hardware reasonably well in the current technologies.

Consider the following example.

```
   ld [%r1+%r5], %r8
      /* r8 := x[i] */
1: umul %r4, %r8, %r8
      /* r8 := A * x[i] */
   st %r8, [%r2+%r5]
      /* zl[i] := r8 */
   subcc %r5, 4, %r5
   bpos,a 1b
   ld [%r1+%r5], %r8
      /* delay slot */
```

There are two principal sources of pipeline stalls in processors without context switching: the first corresponds to the memory access instructions (LD, ST) on dcache miss; in this case the length of the stall is not fixed, and depends on the latencies of the memory subsystem. The second corresponds to the long-latency arithmetic instruction (umul), which has a fixed latency of 4 clock cycles.

To speed up the program, on the assumption we have many threads ready for computation, we can eliminate pipeline stalls by switching the context whenever we detect any of these instructions. The code then may look like this:

```
   ld [%r1+%r5], %r8 ; swch
      /* r8 := x[i] */
1: umul %r4, %r8, %r8 ; swch
      /* r8 := A * x[i] */
   st %r8, [%r2+%r5] ; swch
      /* zl[i] := r8 */
   subcc %r5, 4, %r5
   bpos,a 1b
   ld [%r1+%r5], %r8 ; swch
      /* delay slot */
```

In this code we have explicitly specified where the context switch is to occur. The inefficiency of this approach is in unnecessary context switching, e.g. when the LD or ST instruction generates a cache hit (i.e. the required data are available immediately). Therefore, we would like the instruction pipeline to switch context autonomously and only in the cases when it is necessary (e.g. dcache miss or other unsatisfied data dependencies between instructions). The code would then look the same as the previous code without the *switch* instructions, and context switch will eventualy occur when executing the *umul* or *st* instructions as these require data in register *%r8* that may not be ready at the time of the (first) execution of each instruction. As there are no data dependencies between subsequent iterations of the loop, the loop can be unrolled and treated as a set of independent computing threads that together carry out the operation:

```
f1_start:
   ld [%r1+%r5], %r8
      /* r8 := x[i] */
   umul %r4, %r8, %r8
      /* r8 := A * x[i] */
   st %r8, [%r2+%r5]
      /* zl[i] := r8 */
```

Notice that the loop control instructions have disappeared as these are autonomously executed in hardware by the thread scheduler. Before these threads can execute, the scheduler must be initialized so that it knows how many threads it should create and how registers are to be allocated to the threads:

```
ut_main:
   allocate %r1
      /* FID allocation */
   setstart %r1, 0
   setlimit %r1, MAXLEN-1
      /* family setup */
   setstep  %r1, 1
   setblock %r1, BLOCKSIZE
   set f1_start, %r3
   setthread %r1, %r3
   create %r1, %r2
      /* family creation */
   ...
   mov %r2, %r3
      /* sync on termination */
```

The hardware requirements of microthreading are: use of a self-synchronizing register file (i-structures, [9]), register states to be managed autonomously in the register file, pipeline stalls prevented by context switch in hardware, and thread status and context switch managed autonomously in a hardware thread scheduler.

The micro-threading support on the machine level is represented by the following instructions:

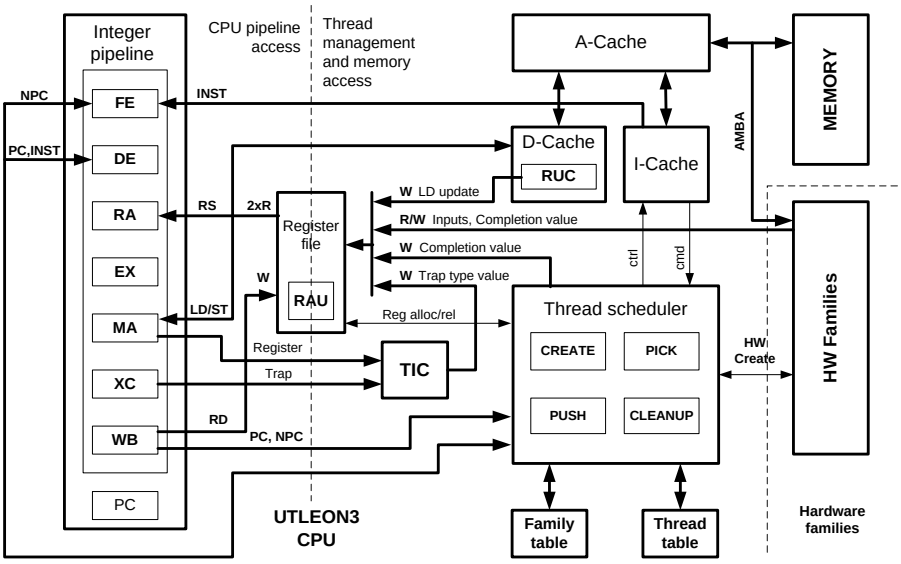

Fig. 1. Architecture of UTLEON3. Pipeline stages: FEtch, DEcode, Register Access, EXecute, Memory Access, eXCeption, WriteBack. RUC - register update controller, RAU - register allocation unit, TIC - trap and interrupt controller.

- **launch** - switches the processor from the legacy mode (user or protected) to the microthreaded mode.
- **allocate** - allocates a family table entry, needed to create a family of threads.
- **setxxxxx** - fills in the allocated family table entry with parameters required by the *create* instruction.
- **create** - creates (a family of) threads based on a family table entry.
- **.registers** - a pseudoinstruction that specifies the number of registers needed by a thread.

Furthermore, each 32-bit instruction word is extended by another two bits that act as an instruction for thread scheduling. Valid combinations are:

- **cont** - continue thread execution,
- **swch** - switch the context to another thread, e.g. on memory load to prevent possible pipeline stall,
- **end** - end thread execution, i.e. the thread ends at this instruction.

The format of assembly instructions has been extended by a field delimited by a semicolon that may contain an explicit instruction for the scheduler. If the field is missing, *cont* is assumed by default.

```
clr %r2
ld [%r1 + %g0], %r3 ; swch
add %r3, %g0, %r4 ; end
```

To keep the 32-bit organization of the memory system in SPARCv8 2-bit extensions for groups of 15 instructions are grouped in one 32-bit instruction word that is located at the beginning of each cache line. One cache line is

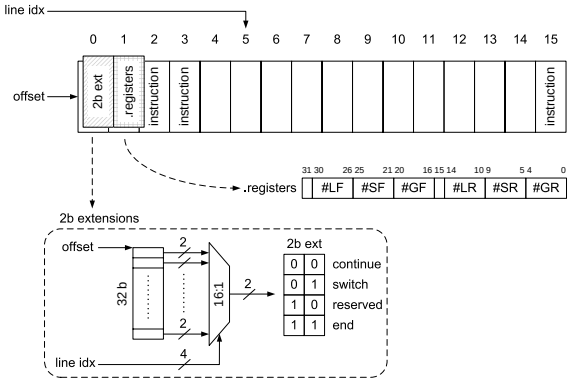

Fig. 2. Organization of the instruction cache. 16 words = 1 cache line

formed by 16 words. The first word of each cacheline is skipped in the micro-threaded mode (explained later in the text). The organization of one instruction cache line is shown in Figure 2.

Micro-threading relies on the use of a self-synchronizing register file based on *i-structures* [9]. To implement the i-structures each register has to be extended to contain the state of its value. A register can be

- **empty** - on power-on reset,
- **pending** - a memory load operation has been requested and no thread has accessed the register since,
- **waiting** - a memory load operation has been requested and a thread has accessed the register since,
- **full** - the register contains valid data.

In the micro-threading model a pending register can be accessed by at most one thread - either by the thread that initiated the pending data update, or by its direct

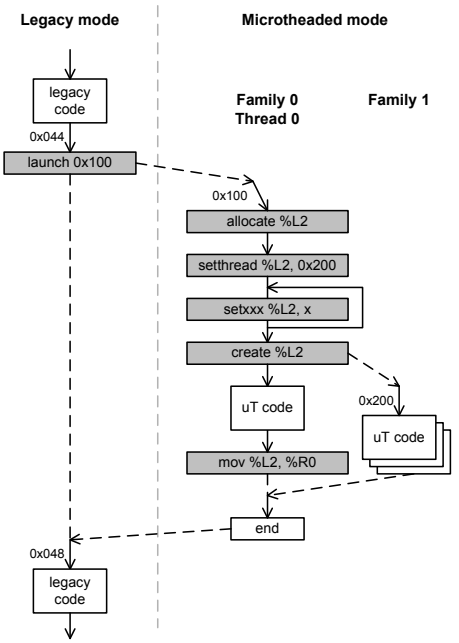

Fig. 3. Program flow.

sibling (only unidirectional data dependencies between direct sibling threads are allowed in micro-threading).

A sample program execution is shown in Figure 3. The processor starts in the legacy mode on power-on reset, then it switches to the microthreaded mode. The parent thread gets synchronized with the children threads by reading the register $\%l2$. On completion of all microthreads the processor switches back to the legacy mode.

III. UTLEON3 Architecture

Figure 1 shows the architecture of UTLEON3, an extended LEON3 with ISE for micro-threading. We have maintained full backward compatibility with LEON3. The core is a 32-bit integer pipeline that executes all legacy instructions. Thread management is implemented in a thread scheduler, which can be seen as a simple 2-bit processor. The instruction word of UTLEON3 is 34 bits wide. All registers have been extended by 2 bits that capture register states, each register is 34 bits long.

A family of threads can be executed either in software, or in a hardware accelerator; this is managed transparently by the thread scheduler without any influence on the coding of the program to be executed. More details on hardware families can be found in [10].

A. Cache Controllers

Load and store requests do not block the integer pipeline. Requests are queued and executed when the corresponding cache line fetch completes.

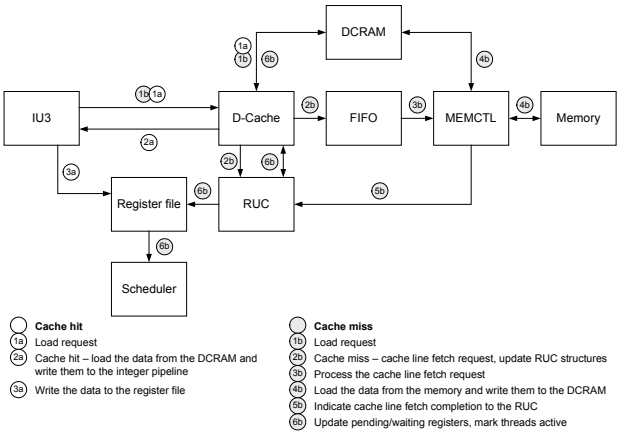

Fig. 4. Data cache hit/miss.

Memory accesses are decoupled from the integer pipeline. The cache controllers are divided in two parts connected through cache line fetch request FIFOs. The pipeline side cache controllers store fetch requests in the FIFOs. The memory side cache controllers process the queued requests. On completion of an instruction cache line fetch all threads waiting for the cache line are marked as ready for execution in the scheduler (put in the *active* queue). Cache lines that are used by threads are locked to prevent their eviction and guarantee forward progress.

On completion of a data cache line fetch all registers that have been waiting for the data in the cache line are updated by the register update controller (RUC). Data cache line fetch scenarios are shown in Figure 4. Instruction cache line misses are handled in a similar manner, more details can be found in [11].

B. Thread Scheduler

The thread scheduler manages the family and thread tables, creates threads, switches context and cleans up the tables on thread completion (see Figure 1). Dynamic register allocation is performed on thread creation by the register allocation unit (RAU). Family table and thread table store information on threads being processed in the processor. Context switch can be the result of an explicit *swch* or *end* instruction, an instruction cache miss or it can occur on reading a register not marked *full*. Threads can be in one of six states; the state transition diagram is shown in Figure 5.

IV. Implementation Results

We have implemented and tested the designed architecture in the Xilinx XUP-V2Pro development board with the XC2VP30 FPGA and in XUP-V5 board with the

TABLE I

FPGA SYNTHESIS - DISTRIBUTION OF FPGA RESOURCES AMONG LEON3 MODULES

Resource type	LEON3s	CACHE	IU3	RF
Slice Flip Flops	1324	246	969	0
Total 6 input LUTs	4804	1651	2805	8
used as logic	4765	1651	2766	8
used as shift registers	39	0	39	0
used as RAMs	0	0	0	0
BRAMs 36kb	7	5	0	1

TABLE II

FPGA SYNTHESIS - DISTRIBUTION OF FPGA RESOURCES AMONG UTLEON3 MODULES

Resource type	UTLEON3s	UTCACHE	UTIU3	UTRF	FTT	TT	SCHED
Slice Flip Flops	4874	2123	1478	292	31	0	850
Total 6 input LUTs	12413	6204	3809	604	58	8	1880
used as logic	12084	6176	3694	554	38	8	1778
used as shift registers	137	0	115	0	0	0	0
used as RAMs	192	28	0	50	20	0	94
BRAMs 36kb	29	13	0	1	7	7	1

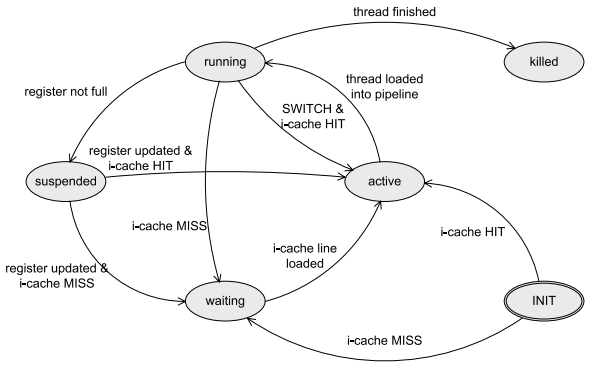

Fig. 5. Transitions between thread states.

XC5VLX110T FPGA. The following data are for the implementation in Xilinx5.

Implementation results are shown in Table I for LEON3 and Table II for UTLEON3. The columns *LEON3* and *UTLEON3* compare complete systems with a processor, 1kB ROM, 4kB RAM and UART. The remaining columns show resource requirements of both legacy blocks (e.g. *CACHE*) and the micro-threaded blocks (e.g. *UTCACHE*). IU3 - integer pipeline, RF - register file, FTT - family thread table, TT - thread table, SCHED - thread scheduler.

LEON3 was configured with 8 register windows, cache associativity 1, cache set size 1kB, cache line size 8W (maximal allowable value for LEON3), and with 136 registers (the standard RISC regfile size).

UTLEON3 was configured with 8 register windows, cache associativity 1, cache set size 1kB, cache line size 16W, family table size 8 items, thread table size 64 items, and with 256 registers.

In contrast to the original LEON3 core, the highest number of resources were consumed by the cache subsystem in the UTLEON3 core – about 50% of all resources in terms of 6-intput LUTs, 44% in terms of flip-flops and 45% in terms of block RAMs.

Both processor cores were implemented in FPGA on Xilinx XUPV5 board in order to verify correct function of the cores and evaluate maximal clock frequency. The maximal frequency was 110MHz in case of the original LEON3 core, and 33.3MHz in case of the new UTLEON3 core.

A. Running Programs in Hardware

The design is downloaded in the board using IMPACT and operated using the Aeroflex-Gaisler GRMON tool. Classical or microthreaded programs are compiled with an extended version of the GNU binutils tools, and either put in the ahbrom.vhd file and synthesized in a ROM, or downloaded as ELF files with GRMON.

As the current version of GRMON does not support UTLEON3 debugging, programs cannot be stepped or stopped once their execution starts, but the instruction and bus trace history shows a (limited) execution history of the user microthreaded program. Program results can be inspected in the memory. Performance data can be read from performance counters that measure specific events in the system (e.g. overall clock cycles, cache miss count, pipeline idle time).

V. PERFORMANCE OF FIR IN UTLEON3

To compare the performance of the modified UTLEON3 pipeline we have implemented and executed a simple

FIR filter both in the original LEON3 processor and the newly developed UTLEON3 processor. The UTLEON3 execution either makes or does not make use of the hardware families of threads.

$$z_k = \sum_{i=0}^{L-1} b_i x_{k+i}$$

The L parameter specifies the length of the filter. In typical embedded DSP applications (echo cancellation, ADSL) the length is often of the order of tens of elements (taps); we can safely assume here a typical $L = 32$.

```
int x[N], z[N], b[L];
for (k = 0; k < N-L; k++) {
  int s = 0;
  for (i = 0; i < L; i++) {
    s += b[i] * x[k+i];
  }
  z[k] = s;
}
```

```
int x[N], z[N], b[L];
F1: create_family (index int k;
                   start=0; limit=N-L-1) {
  int s = 0;
  F2: create_family (index int i;
                     start=0; limit=L-1
                     global={k}; shared={s}) {
    s += b[i] * x[k+i];
  }
  z[k] = s;
}
```

Fig. 6. A complete FIR program; left - legacy, right - microthreaded

Figure 6 shows a complete software FIR program that computes the filter equation over an array of N elements. The inner family F2 implements the filter equation by unrolling the sum into a family of dependent threads (using a shared integer s). These threads are coupled by the addition operator (line $s+ = b[i] * x[k+i];$) which is both associative and commutative.

The *create-family* pseudo-command from the code example is further decomposed into a sequence of assembly-level instructions: **allocate**, **setstart**, **setlimit**, **setstep**, **setthread**, and **create**. These instructions will be assigned their parameters from the arguments of the *create-family* pseudo-command. Besides the obvious parameters (*index*, *start*, *limit*, *step*), which directly correspond to a classical *for*-loop construct, there are some other that need an explanation: *global*, *shared*, and *blocksize*.

The *global* and *shared* parameters specify lists of variables (registers) that will be made visible to the family being created. The difference between these two is that the *global* ones stay fixed during the course of execution of the family, while the *shared* ones are assumed to be passed–and possibly modified–from one thread to another. This sharing of data is strictly unidirectional and always only between two adjacent threads in the family, i.e. from a thread indexed i to a thread indexed $i + 1$. Global and shared variables are directly supported by the machine architecture by the means of thread global and shared registers. The quantity of these registers can be individually customized for each family using the **.registers** assembly directive.

In the benchmark example the *global* parameter is used to specify that variables A and Y will be passed to the thread family.

The final family parameter to be described is the *blocksize*. This parameter is optional for it does not affect the *semantics* of the computation, but it affects its pace. The *blocksize* specifies the maximal number of threads of a given family that are allowed to co-exist at any moment. This enables a compiler or assembly-level programmer to artificially throttle the rate of thread creation so as not to congest the memory subsystem or the underlaying large register file from which the registers are dynamically allocated. Also, as the processor does not implement virtualization of some internal data structures yet (notably the *Family Table* and *Thread Table*), we are bound to use the *blocksize* parameter to prevent certain families from consuming all available internal resources.

The FIR filter examples with various unroll factors were compared. All versions are equivalent in terms of generated data outputs. The higher unroll factor should reduce runtime in both legacy mode and microthreaded mode. In the case of legacy mode the higher unroll factor allows better arrangement of instructions, as was mentioned above. In case of microthreaded mode the higher unroll factors results in longer threads, which reduces load of a thread scheduler.

The results are shown in Table III (also see Fig. 7). Examples with unroll factor one, two and four were used for both the legacy code (marked as L3 FIR 1x, L3 FIR 2x, L3 FIR 4x respectively) and the microthreaded code (marked as UT FIR 1x, UT FIR 2x, UT FIR 4x respectively). Structure of the table and the graph is the same as in the previous experiment.

As expected, the number of pipeline stalls due to internal data dependences (see column "Stalls") was smaller in versions with higher unroll factors. Nevertheless the reduction due to loop unrolling was noticeably smaller when compared to the reduction due to application of microthreading. Considering the UTLEON3 processor, the application of microthreading reduces the number of stalls 35 times on average – while it represented 43% of the overall runtime on average in case of the legacy code, it was just about 2% of the overall runtime in case of the microthreaded code.

The higher unroll factor results in lower number of stalls due to thread management (see column "UT Overhead"). This overhead is 38% in case of unroll factor 1 (UT FIR 1x), and decreases down to 5% in case of higher unroll factors (UT FIR 2x, UT FIR 4x).

A further, more general analysis of speed-up limits of UTLEON3 for pipelined and non-pipelined long-latency operations can be found in [12].

TABLE III
PIPELINE EXECUTION PROFILE FOR A 26-TAP FIR FILTER

Core	Program	Total	Working		Holdn(cache)		Stall		UT overhead	
	[1]	[1]	[1]	[%]	[1]	[%]	[1]	[%]	[%]	
LEON3	L3 FIR 1x	8976	4873	54	231	3	3872	43		
UTLEON3	L3 FIR 1x	9703	4873	50	190	2	4640	48		
UTLEON3	UT FIR 1x	5877	3506	60	16	0	107	2	2248	38
LEON3	L3 FIR 2x	8230	4873	59	253	3	3104	38		
UTLEON3	L3 FIR 2x	8189	4873	60	212	3	3104	38		
UTLEON3	UT FIR 2x	4612	4274	93	12	0	104	2	222	5
LEON3	L3 FIR 4x	7477	4105	55	268	4	3104	42		
UTLEON3	L3 FIR 4x	7421	4105	55	212	3	3104	42		
UTLEON3	UT FIR 4x	4210	3890	92	14	0	98	2	208	5

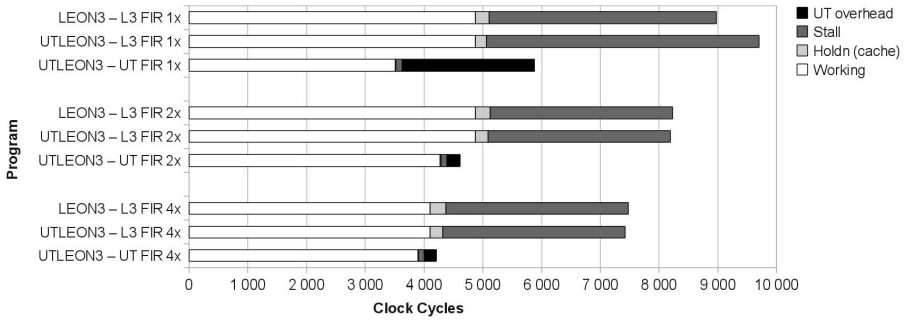

Fig. 7. Pipeline execution profile for a 26-tap FIR example

VI. UTLEON3 VS. OPENSPARC T1/T2

The OpenSPARC T1 and T2 processors are open-source implementations of the UltraSPARC 2005 and 2007 architectures [1]. OpenSPARC T1 has been ported to Xilinx FPGAs; in the Xilinx XUPV5-LX110T Evaluation Platform T1 has been reported to operate at 62.5MHz in a 125MHz MicroBlaze system [13]. As the UTLEON3 processor is based on the SPARC v8 ISA it is only natural to compare it against the OpenSPARC processors.

UltraSPARC 2005, 2007 [1] architectures include support for Chip-Level Multithreading (CMT). The purpose of the CMT is to define multi-processing interface between the software and hardware. In CMT a processor (physical module which plugs into a system interconnect fabric) is a collection of physical cores. Physical core is a pipeline with caches and other associated hardware. One or more software threads - called strands - may be scheduled on one physical core. A strand is the software-visible state (PC, NPC, GP and FP registers, ASRs etc.) that the hardware must maintain in order to execute a software thread. From the ISA point of view the strand behaves like a virtual processor including its own MMU context. Therefore, the recommended programming model for CMT processors is either Posix Threads (pthreads) or OpenMP, which are both well-known industry standards. The incentive is to offer coarse-grained parallelism for task-level multitasking.

Contrary to that, the micro-threaded concurrency model employed in UTLEON3 is fine-grained. The goal of micro-threading is to extract instruction-level parallelism from existing sequential algorithms [14].

VII. CONCLUSIONS

This paper has described an implementation of instruction set extensions for micro-threading in SPARC. The architecture of key functional blocks of the UTLEON3 processor have been presented together with implementation data for Xilinx XC5VLX110T FPGA. The speedup of micro-threading in UTLEON3 over identical programs in LEON3 has been shown and discussed. The final development of UTLEON3 will be made available to the research community at [15].

ACKNOWLEDGMENT

This work was supported and funded by the European Commission under Project Apple-CORE No. FP7-ICT-215215, and by the Czech Ministry of Education under Project No. 7E08013. The paper reflects only the authors' view; neither the European Commission nor the Czech Ministry of Education are liable for any use

that may be made of the information contained herein. For information about the Apple-CORE project see [15].

REFERENCES

[1] T. Takayanagi, J. L. Shin, B. Petrick, J. Su, and A. S. Leon, "A dual-core 64b ultrasparc microprocessor for dense server applications," in *DAC*, S. Malik, L. Fix, and A. B. Kahng, Eds. ACM, 2004, pp. 673–677.

[2] P. Kongentira, K. Aingaran, and K. Olukotum, "Niagara: a 32-way multithreaded SPARC processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

[3] K. D. Kissell, "MIPS MT: A multithreaded RISC architecture for embedded real-time processing," in *HiPEAC*, ser. Lecture Notes in Computer Science, P. Stenström, M. Dubois, M. Katevenis, R. Gupta, and T. Ungerer, Eds., vol. 4917. Springer, 2008, pp. 9–21.

[4] J. Gaisler, E. Catovic, and S. Habinc, *GRLIB IP Library User's Manual*. Gaisler Research, 2007.

[5] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[6] C. R. Jesshope and B. Luo, "Micro-threading: A new approach to future RISC." in *Proceedings of the 5th Australasian Computer Architecture Conference*. IEEE Computer Society press, 2000, pp. 34–41.

[7] C. Jesshope, "Scalable instruction-level parallelism," in *Computer Systems: Architectures, Modeling, and Simulation*. Springer Berlin / Heidelberg, 2004, pp. 383–392.

[8] C. R. Jesshope, "muTC - an intermediate language for programming chip multiprocessors," in *Asia-Pacific Computer Systems Architecture Conference*, 2006, pp. 147–160.

[9] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transaction on Computers*, vol. 39, no. 6, pp. 300–318, 1990.

[10] J. Sykora, L. Kafka, M. Danek, and L. Kohout, "Microthreading as a novel method for close coupling of custom hardware accelerators to SVP processors," in *Proceedings of the 14th EUROMICRO Conference on Digital System Design (DSD2011)*. Conference Publishing Services, 2011.

[11] M. Danek, L. Kafka, L. Kohout, and J. Sykora, "Instruction set extensions for multi-threading in LEON3," pp. 237–242.

[12] J. Sykora, L. Kafka, M. Danek, and L. Kohout, "Analysis of execution efficiency in the microthreaded processor UTLEON3," in *Proceedings of the 2011 Conference on Architecture of Computing Systems (ARCS 2011)*, ser. Lecture Notes in Computer Science, vol. 6566. Springer, 2011, pp. 110–121.

[13] Sun Microsystems. RAMP retreat August, 2008 update. http://www.opensparc.net/publications/presentations/ramp-retreat-august-2008-update.html.

[14] C. R. Jesshope, J.-M. Philippe, and M. van Tol, "An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the svp model of concurrency," in *SAMOS*, ser. Lecture Notes in Computer Science, M. Berekovic, N. J. Dimopoulos, and S. Wong, Eds., vol. 5114. Springer, 2008, pp. 218–228.

[15] The Apple-CORE Consortium. Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs. http://www.apple-core.info.