Microthreading as a Novel Method for Close Coupling of Custom Hardware Accelerators to SVP Processors

Jaroslav Sykora, Leos Kafka, Martin Danek, and Lukas Kohout.
Department of Signal Processing
Institute of Information Theory and Automation of the ASCR (UTIA AV CR, v.v.i.)
Pod Vodarenskou vezi 4, Prague, Czech Republic
Email: {sykora, kafkal, danek, kohoutl}@utia.cas.cz

***Abstract*—We present a new low-level interfacing scheme for connecting custom accelerators to processors that tolerates latencies that usually occur when accessing hardware accelerators from software. The scheme is based on the Self-adaptive Virtual Processor (SVP) architecture and on the micro-threading concept. Our presentation is based on a sample implementation of the SVP architecture in an extended version of the LEON3 processor called UTLEON3.**

The SVP concurrency paradigm makes data dependencies explicit in the dynamic tree of threads. This enables a system to execute threads concurrently in different processor cores. Previous SVP work presumed the cores are homogeneous, for example an array of microthreaded processors sharing a dynamic pool of microthreads. In this work we propose a heterogeneous system of general-purpose processor cores and custom hardware accelerators. The accelerators dynamically pick families of threads from the pool and execute them concurrently. We introduce the Thread Mapping Table (TMT) hardware unit that couples the software and hardware implementations of the user computations. The TMT unit allows to realize the coupling scheme seamlessly without modifications of the processor ISA.

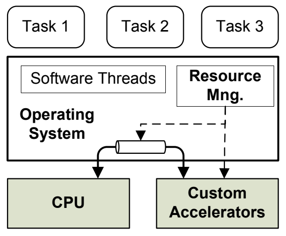
The advantage of the described scheme is in decoupling application programming from specific details of the hardware accelerator architecture (identical behaviour of a software create and hardware create), and in eliminating the influence of hardware access latencies. Our simulation and FPGA implementation results prove that the additional hardware access latencies in the processor are tolerated by the SVP architecture.

***Keywords*-Microthreading, SVP concurrency model, custom accelerators, hardware families of threads, UTLEON3 processor.**

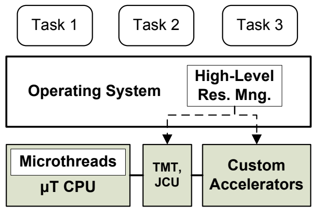
I. Introduction

Processor architectures are progressively evolving towards multicores. The scaling of the manufacturing process makes placing many cooperating processor cores on a chip feasible. An array of processing cores could also simplify the energy and performance scaling, were it carried out on-line or during the system-on-chip design.

Figure 1 shows organization of a multithreaded system composed of both the general-purpose and specialized processors (custom accelerators). In the traditional organization (Figure 1a) threads are managed and scheduled by the operating system (e.g. a pthreads library on top of Linux). This is true even for many processors that directly support multiple threads executed in hardware for they use the same software abstraction layers anyway. In this organization accelerators can be managed by middleware software in the operating system, which can also take care of any context-switch mechanisms they require to support multithreaded execution. However, in the **micro**threaded environment (Figure 1b) the thread execution is exposed to application programs at the processor *Instruction Set Architecture* level. This organization calls for a novel interfacing scheme between the general-purpose and specialized processors that can exploit the parallelism of the SVP model.



(a) Custom accelerators managed by OS with software threads.



(b) Accelerators in the microthreaded environment.

Figure 1. Comparing multithreaded systems: traditional and microthreaded organizations.

The structure of the paper is as follows: In this section we overview the previous related work, and state the contributions of this paper. In Sections II and III the reader is introduced to the SVP model and the UTLEON3 processor microarchitecture, respectively. In Section IV the proposed microthreaded coupling scheme is presented. In Section V experimental evaluation is given, and we conclude the paper in Section VI.

A. Low-level Accelerator Coupling Schemes

From the hardware organization perspective the custom accelerator can be situated relative to the processor in three main positions [1]. In the **Attached Processor** scheme the accelerator is placed on an I/O bus, along with other peripheral modules; e.g. *BCE*[1] [2], *Morpheus* [3]. Memory mapped registers or local memories are typically used as a communication interface, and synchronization is handled in software by polling or by interrupts. Attached processors do not depend on the controlling processor ISA (*Instruction Set Architecture*), and their I/O interface operates on well-defined system buses; thus they are easily portable. However, as the application and configuration data have to be transferred explicitly to/from them by the CPU, and as the synchronization is handled in software, the latencies and control overhead can be high; thus greater parts of a computation need to be implemented in the accelerator to take advantage of data locality and to combat the overhead.

In **Reconfigurable Functional Units (RFU)** the reconfigurable logic is an integral part of the processor pipeline, along with other functional units, and instructions are issued into and retired from the RFU using the same mechanism; e.g. *Chimaera* [4], *Garp* [5].

[1]BCE = Basic Computing Element

Data are communicated implicitly through the processor register file. The RFU hardware is not portable as it is an integral part of the host processor and its ISA. For the scheme to be effective the compiler for the processor architecture has to be extended to automatically analyze the application source code, identify parts suitable for RFU execution, generate configuration bitstreams for RFU, and use the newly created instructions during the code generation process [1].

In the **Coprocessor** scheme the accelerator has an independent processing pipeline and register set; e.g. *MOLEN* [6]. The crucial difference from the *attached processors* scheme is that the synchronization between the CPU pipeline and the coprocessor pipeline is handled in hardware, thus reducing control overheads. For example, a coprocessor can stall the CPU pipeline when it executes a coprocessor instruction while the previous one has not completed yet.

Multithreaded architectures in general introduce a problem of resource sharing of the reconfigurable accelerators between multiple threads [7]. At the lowest level this is manifested by the need for a synchronization and a context switch mechanism. In RCC [8] (multithreaded extension of the Molen [6] protocol), only one thread at a time is allowed to execute in a reconfigurable unit. In Chimaera [4] the RFU is always stateless (i.e. it is purely combinatorial) so there is not any context to be saved on a thread switch. On the other hand, the Garp [5] RFU architecture provides special instructions for context save/restore on a thread switch to be used by the operating system.

B. Contributions of the Paper

The **Microthreaded Coupling Scheme**, proposed in this paper, uses hardware synchronization between the processor's *Thread Scheduler* and the subsystem of custom accelerators. Parameters are communicated through the processor *Register File* the same way as in a normal function call. However, the custom accelerators themselves are *not* tailored specially for our coupling scheme: in the case study we used our in-house framework that was designed for a traditional single-threaded system with the *MicroBlaze* processor [2].

We implement a **Thread Mapping Table** (TMT, see below), a special hardware unit which enables the coupling scheme to be implemented *without* modifications to the SVP processor ISA (such as introducing a new instruction, *cf.* the MOLEN protocol [6]).

C. Other Related Work

The GPGPU[2] programing model is multithreaded, however the underlying architecture executes threads in lock-step in SIMD blocks, called *warps*, that typically comprise 16-32 threads. The system delivers high performance only when all threads within one *warp* execute the same instructions. The low-level interfacing scheme is the *attached processor*. By contrast the SVP model is targeted to general-purpose programs.

BORPH [9] is an operating system based on Linux that was extended to support reconfigurable hardware accelerators as the first-class citizens in the environment. The operating system treats accelerators as stand-alone processes, and provides them unified file I/O and resource management services (Figure 1a).

[2]GPGPU = General-purpose computing on graphics processing units

When partitioning an application, its high-level control functions are implemented in software and the low-level 'number-crunching' functions are moved to custom hardware. A common problem is to determine the right level for the decomposition. In this regard one extreme is the *reconfigurable functional unit* that inserts fine grained custom hardware directly into the processor data path. The other extreme is the BORPH OS which treats custom hardware as an OS-level process.

II. SVP Concurrency Model

The *Self-adaptive Virtual Processor* (SVP) is a general concurrency model used to design and program multithreaded multicore systems. It has been described in many previous publications [10], [11].

The SVP model expresses fine-grained concurrency by composition of *microthreads*.[3] A microthread comprises only a few processor instructions that typically implement a body of a loop and share only a small portion of the processor register file. A *family of threads* is an ordered set of threads, all created by one processor instruction (called `create`). The ordering is defined by a sequence of integer index values that are specified during the create event as a {start, step, limit} triple. These values are provided by the program using the `setstart`, `setstep`, and `setlimit` instructions, respectively. Microthreads may execute in parallel by default, and only explicit unidirectional dependencies between successive threads within one family, and between a parent thread and its child family, are allowed.

By creating a family of many fine-grained threads in one event the SVP model allows an implementation to amortize overheads associated with individual thread management. A family of threads represents a batch of coarse-grained work that is to be scheduled in a multicore system. In contrast to the GPGPU model, the individual threads are executed independently so that no restriction is placed on conditional branches in threads.

To ensure an acyclic dependency graph of threads, the SVP model allows only unidirectional dependencies between microthreads. This enables a processor implementation to freely choose the number of concurrent threads it will schedule together, called the *blocksize* parameter (provided the implementation reserves resources for microthreads as per their ordering within the family to avoid the deadlock).

A. Multiple Cores in SVP

The SVP model is oblivious to the number of processing cores in the system. A given family of threads can be dynamically scheduled entirely on a single processor, or distributed across multiple cores. In the latter case each thread executes in one given core, but different threads can be scheduled in different cores. Each family of threads is associated to a compute cluster (called 'local place') composed of several tightly coupled cores. Threads of the family are scheduled concurrently within their local cluster. To delegate a family of threads to another cluster the programmer can use the `setplace` instruction.

The previous SVP work assumed that cores within clusters are homogeneous (at least from the ISA point of view) to enable

[3]The terms 'thread' and 'microthread' are used interchangeably throughout the text.

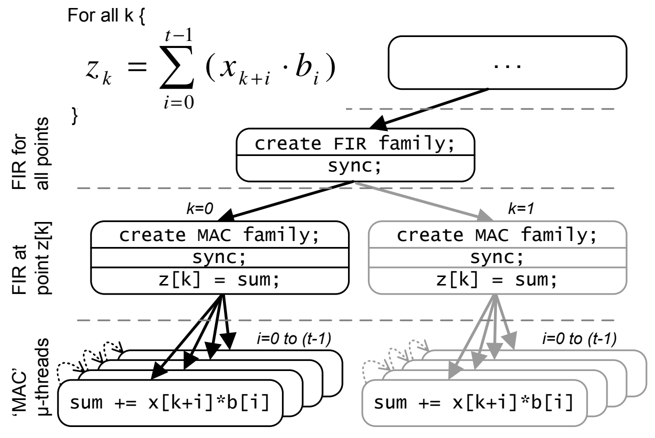


Figure 2. Graphical representation of (a part of) the microthreaded FIR computation with nested families of threads. A tree of threads is dynamically created. The arrows represent the dependencies visible in the SVP model.

transparent thread migration. In this work we envision heterogeneous microthreaded clusters composed of both the general-purpose and specialized processors (custom accelerators). As the specialized processors by definition cannot execute all kinds of threads, the system requires additional information to discriminate among them so that they can be distributed correctly in the cluster. Instead of extending the SVP ISA with a new instruction, we introduce a *Thread Mapping Table* (TMT) hardware unit to provide the information. Conceptually the TMT unit is a software-controlled associative table that identifies threads by their memory addresses and maps them to a class of specialized processors which can execute the threads more efficiently than the general-purpose core. The table is to be administered by a resource management software based on information provided by applications, and with respect to the current configuration of the specialized processors (if reconfigurable accelerator cores are used). In the microthreaded coupling scheme we further take advantage of the fact that threads are created in groups (families) to amortize the cost of processing in the TMT unit. Only the family creation event itself has to be routed through the TMT unit, not the individual microthreads.

B. Program Example

Figure 2 shows an implementation of a FIR[4] filter that we use to demonstrate the approach. The filter output is defined by the equation:

$$z_k = \sum_{i=0}^{t-1} (x_{k+i} \cdot b_i) \qquad (1)$$

where z is an output vector (with $n-t+1$ elements), x is an input vector (with n elements), b is a vector of coefficients (taps), and t is the length of vector b. We assume $n \geq t$.

In a sequential program the computation can be implemented using two nested *for* loops: the inner loop computes the dot product, while the outer loop runs over the whole index space of the vector z. In the SVP paradigm we implement both loops as nested families of microthreads, represented by oval boxes in Figure 2.

[4]FIR = Finite Impulse Response

The hardware-accelerated version of the compute kernel itself was developed using the approach described in [2]: Custom accelerator cores, called *BCE*, are reconfigurable vector processing units specialized for a limited set of operations that are required in a given application. All BCEs are interfaced by dual-ported (DP) memory blocks that are common in FPGAs. The input data (arrays x and b) are loaded into two local memory blocks, the FIR BCE is configured, the computation is executed, and finally the output data is unloaded from the third memory block to array z. The data transfers are *not* handled by the accelerators themselves. The BCE cores are deliberately *not* aware of the microthreaded execution in the main processor, thus existing accelerators can be reused.

III. Microthreaded UTLEON3 Processor

The *UTLEON3* processor is an implementation of the SVP model for managing concurrency. The *UTLEON3* core is written in VHDL, it has a 7-stage in-order single-issue multi-threaded execution pipeline, and it is fully FPGA-synthesizable. It is based on the *LEON3* SPARCv8 embedded processor from *Aeroflex Gaisler*. For more information please see the prior work [12]; the section below summarises the microarchitecture.

A. Microarchitecture Overview

Register File: Traditional multi-threaded architectures often replicate the whole processor state for each thread context supported, including all the architectural (program visible) registers [13], [14]. This simplifies the software programming model, but it requires a large register file to support just few thread contexts. As the processor register file is one of the most expensive units in a CPU, its optimal utilisation is very important. The UTLEON3 processor allows each thread to individually specify the number of required registers, from 1 up to the architectural limit of 32 registers per thread given by the SPARCv8 instruction encoding.

Fine-grained synchronization and communication between threads in one family, and between the processor pipeline and long-latency units (cache, FPU) is accomplished by a self-synchronizing register file. Each 32b register in the file is extended with a state information. A register can be either *FULL* when its data is valid, or *PENDING* when its data is scheduled to arrive, or *WAITING* if there is a thread waiting for the data.

Thread Scheduling: The processor implements blocked (coarse) multithreading, meaning a thread is switched out of the pipeline only when an unsatisfied data dependency (i.e. a dependency on a long-latency operation) has been encountered. This improves the single-thread performance, but it requires the pipeline to have fully bypassed stages. The processor allows instructions from different threads to be present in distinct pipeline stages at the same time.

B. Previous Multithreaded Processor Architectures

Multicore architectures require many concurrent tasks to be fully utilized. Existing architectures usually presume a multi-programmed environment (e.g. *OpenSPARC T1/T2*), where processes communicate through shared memory or by message passing. Others provide ISA extensions to support multiple thread execution [15], but they do not automatically spread threads over multiple cores. The *MIPS MT* architecture introduces only two new unprivileged instructions: `fork` to create a new thread, and `yield` to make a thread wait for an event. The *Sparcle/Alewife* [13] system is based on a slightly modified SPARC architecture. It employs block (coarse) multithreading. The SPARC register windows are used to implement 4 independent thread contexts rather than as a register stack; context switching and thread scheduling is done in software via fast traps; and fine-grain synchronization through *empty/full bits* is implemented in an external Communications and Memory Management Unit and in the cache controller. The *MSparc* [14] architecture is similar to Sparcle, but the context switching mechanism is provided in hardware.

IV. Description of the Microthreaded Coupling Scheme and Its Implementation

The accelerator subsystem, called HWFAM,[5] is depicted in Figure 3, along with an outline of the UTLEON3 CPU. The HWFAM subsystem is directly connected to the *thread scheduler* and the *register file* (RF) of the microthreaded processor. Note that these connections are mandated by the SVP model which assumes there are communication channels among cores within a cluster to enable the distribution of threads (Section II-A).

A. Family Creation Event

A *create event* in the SVP paradigm represents a family of threads (a batch of work) that shall be scheduled and executed in the system. It is a consequence of executing the `create` instruction that starts the whole family (*not* an individual thread within the family). The create event includes all parameters of the family:

(a) Family ID (*fid*) – uniquely identifies the family instance in the compute cluster.
(b) Thread starting address in the instruction memory – uniquely identifies the *function* of the family of threads.
(c) Start/step/limit values defining the index space of the family of threads – roughly correspond to the *amount of work* in the family.
(d) Base addresses of the family global and shared registers in the processor register file – specifies the register locations of further family parameters (apart from the start/step/limit values).
(e) Address of the synchronization register – for signalling the child family termination back to the parent thread.

B. Thread Mapping Table

All create events that originate in the *thread scheduler* of the CPU are communicated to the *Thread Mapping Table* (TMT) in the HWFAM subsystem (see step 1 in Figure 3). The table maps families of threads implemented in software to their hardware-accelerated counterparts that are functionally equivalent (by definition). In the mapping table the software thread program address (32 bits) is looked up in a hardware associative table.

We implemented the TMT unit as a content-addressable fully-associative memory in hardware (see the discussion in Section V-B). Thread starting addresses from the create events are used as the lookup keys. The actual table contents is loaded during program initialization by software. To this end the table is accessible through the internal configuration bus and the host bridge as a standard memory-mapped device to the main processor.

[5]HWFAM = Hardware Families of Threads

Data transfer and control overheads can render families with small number of threads perform worse in a hardware accelerator than in the general-purpose processor. Using the *start*, *step* and *limit* fields of the create event the TMT unit computes the size of the family of threads: $size = (limit - start + 1)/step$. (In our current implementation we assume $step$ is a compile-time known constant, thus we can avoid the division.)

Therefore when the lookup performed in TMT by the thread address succeeds, the $size$ of the family of threads is further compared to the $minimal$ and $maximal$ sizes specified in the table. Finally, when the decision is made, a response is sent back to the CPU *thread scheduler*; when the family cannot execute in an accelerator (it was not found in the TMT, or the size check has failed) the general-purpose processor continues with a normal execution of the family of threads. Otherwise, the create event is handed over to the *Job Control Unit* (JCU) and the microthreaded processor is informed that it does not have to schedule the execution of the particular family of threads.

C. BCE Cores and Data Transfers

The external interface of a BCE custom accelerator core is very simple. It consists merely of dual-ported local memories that are common in the FPGA technology (Figure 3 right). One memory port of each dual-ported memory is reserved for the core, the other for a master controller, such as the Job Control Unit or the DMA engine. Each BCE core can be connected to several local memories; one memory block is reserved for control and configuration functions.

The organization described eases the achievement of the timing closure, particularly if the BCE core is implemented in a different clock domain, and it also facilitates FPGA partial reconfiguration. More information can be found in [2].

Coarse data transfers between the local memory blocks of BCE accelerator cores and the main memory connected over the AMBA bus are handled by a configurable *DMA Engine*. The DMA engine can be set-up from inside the subsystem by the Job Control Unit because its configuration interface is facing inwards. The *Host Bridge* allows an outside entity (the main processor) to access the subsystem internals during initialization and for debugging purposes.

D. Job Control Unit for Accelerator Control

Sequencing of tasks within the subsystem is handled by the *Job Control Unit* (JCU). It is based on the PacoBlaze 3 microcontroller that implements Xilinx's 8-bit KCPSM3 ISA. We made the following modifications to the PacoBlaze 3 microcontroller to improve its programmability and performance: *(a)* all internal data paths were extended to 16 bits and two new instructions were added to support the 16 b processing; *(b)* the processor I/O port protocol was extended with a hardware busy signal; *(c)* a *register+offset* addressing mode was added to the I/O and scratch-pad access instructions.

The JCU handles all the 'impedance matching' between software calling conventions and execution in accelerators. During system initialization a firmware is downloaded into the JCU's embedded microcontroller. The microcontroller executes an event-driven program in a loop to handle the following tasks:

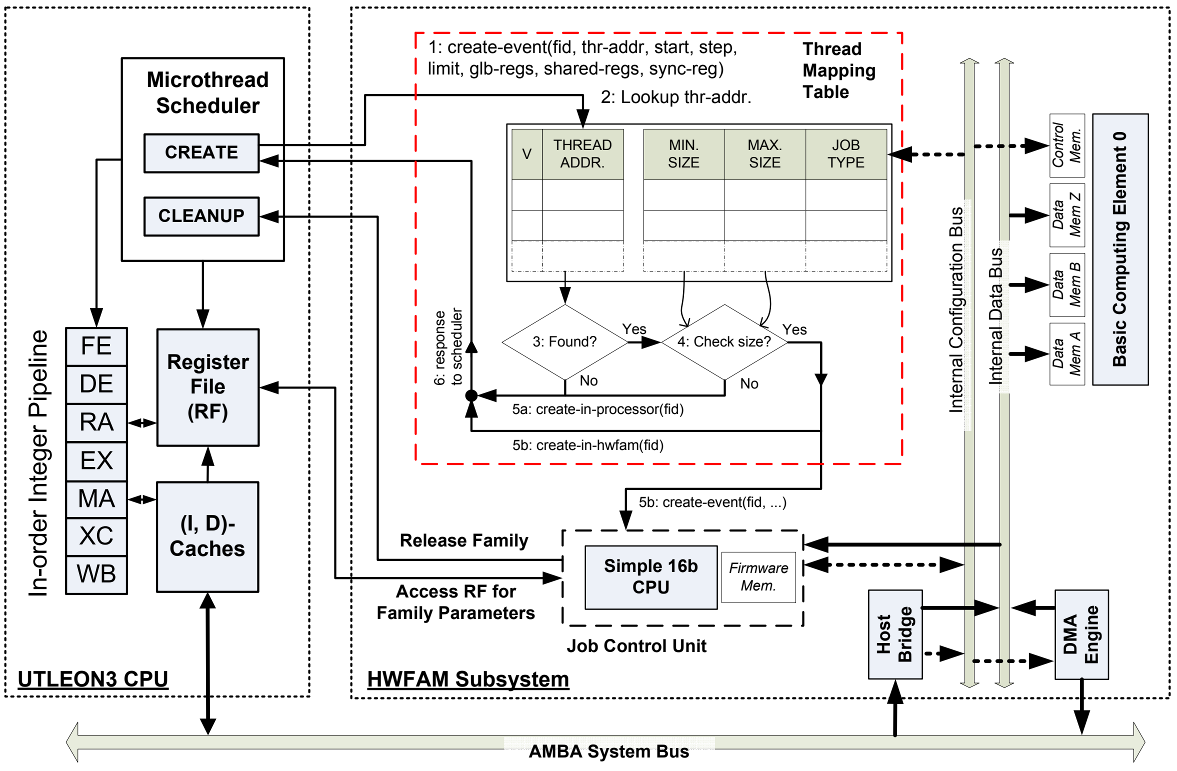


Figure 3. Block diagram of the HWFAM accelerator subsystem, with connections to the UTLEON3 processor (the CPU part of the picture is highly simplified for brevity). The CPU contains an in-order 7-stage integer pipeline, non-blocking caches, self-synchronizing register file, and the microthread scheduler. The HWFAM subsystem is composed of the Thread Mapping Table (TMT) unit which directly interfaces the microthread scheduler and the register file of the CPU, and the Job Control Unit (JCU) that handles the control operations.

- Communication with the TMT unit and UTLEON3 *Thread Scheduler* to receive and release families of threads.
- Access to the UTLEON3 *Register File* to fetch family global and shared registers that are part of the software calling convention. Typically the family global registers contain memory addresses required to configure the DMA engine, and the special *sync* register is used to signal completion of a family of threads.
- Configuration and triggering of channel transfers in the DMA engine and computations in BCE cores.

Thanks to the event-driven programming the JCU can control multiple BCE cores at a time, it supports data streaming with ping-pong buffering, and dynamically overlaps computation and data transfers. This enables acceleration of families of threads that access I/O arrays that are larger than the limited capacity of the BCE local memories allows.

E. Discussion

The previous work, for example the MOLEN protocol [6], utilizes the main processor for accelerator control because the processor is unused anyway during the accelerator run-time. In the microthreaded coupling scheme we introduce an additional simple processor in the *Job Control Unit* to handle the accelerator control because we expect the microthreaded processor to continue executing other unrelated families of threads in the meantime.

The approach also decouples hardware and software application programming and toolchain flows. The application software binary need not be extensively modified to support hardware accelerated execution, and on the other hand the low-level control of custom accelerators can be hidden in the programmable *Job Control Unit*.

In the scheme the JCU autonomously accesses the CPU register file to read parameters of a family of threads stored in its general-purpose registers. This requires an additional port to the register file. However, the UTLEON3 processor already contains such a port, which is used for late (non-critical) updates of the register file from D-Cache. As the HWFAM subsystem accesses the register file only when a family is created and when it finishes, we time-share the existing RF port. Thus we do not introduce any significant additional costs in the processor in terms of resources and clock-cycle timing.

In our scheme we can easily take advantage of the family *start*, *step*, and *limit* parameters that are already available during the create event to dynamically decide if the function (family) is suitable for hardware execution.

As the hardware resource allocation for families of threads is centralized in the JCU, our scheme automatically handles concurrent execution without the need for locking and synchronization between otherwise unrelated threads.

V. Experiments

The FIR program in Figure 2 was implemented in the *mtsparc* assembly for execution in the microthreaded processor UTLEON3.

Table I
SYNTHESIS RESULTS IN *Xilinx Virtex 5* TECHNOLOGY (XC5VLX110T). *r* = NUMBER OF ROWS IN THE ASSOCIATIVE THREAD MAPPING TABLE.

Component	BRAM	FFs	LUTs	DSP48E
UTLEON3	86	5405	10874	4
DMA + Bridge + Internal Bus	15	495	549	0
Job Control Unit	4	126	409	0
FIR BCE	6	216	114	3
DCT BCE	2	542	451	4
Thread Mapping Table:				
r = 16 rows	9	915	644	0
r = 8 rows	8	665	529	0
r = 4 rows	8	539	495	0
r = 2 rows	8	475	455	0

Table II
CPU EFFICIENCY IPC DEGRADATION DUE TO THE ADDED LATENCY OF THE TMT UNIT; n IS FIR INPUT VECTOR LENGTH; NUMBER OF TAPS: $t = 24$.

n	CPU IPC		IPC Degradation
	no TMT	*with TMT*	
26	0.659	0.634	-0.025
32	0.665	0.661	-0.004
40	0.676	0.673	-0.002
48	0.681	0.676	-0.005
64	0.680	0.673	-0.007
80	0.679	0.673	-0.006

Simultaneously an accelerated implementation in a BCE core was developed. The FIR BCE core was instantiated twice in the HWFAM subsystem to allow a concurrent execution of up to two families of threads in hardware. The JCU allocates a FIR BCE for a particular instance of a family of threads dynamically. In our implementation the JCU temporarily switches off the TMT unit when both the BCE cores are occupied. However, other job scheduling schemes can be implemented in the JCU firmware.

In the experiments we chose the number of FIR taps to be constant ($t = 24$) as this is a typical value used in the embedded DSP domain for signal processing tasks.

The whole SoC design (UTLEON3 processor, HWFAM subsystem, peripheral modules) can be synthesised in the *Xilinx XUPV5-LX110T Evaluation Board (ML509)* that is supported in Gaisler's GRLIB package. In the experiments the main system memory had a latency of 2 waiting cycles per word as this is the required configuration of the external on-board 1MB SRAM.

A. Hardware Synthesis Results

The system was synthesized using *Synopsis Synplify D-2010.03* and *Xilinx ISE 12.3* in the *Xilinx Virtex 5* XC5VLX110T part; Table I lists the synthesis results. Apart from the FIR BCE core, a hardware implementation of the *Discrete Cosine Transform* (DCT) from *JPEG* was developed to compare the hardware resource utilization with the other components. The number of rows r in TMT governs the number of classes of families of threads (distinct functions) that can be mapped to hardware accelerators at the same time. In the presented experimental setup with two equivalent FIR BCEs we required only one TMT row that maps the address of the FIR family in the *mtsparc* binary to a FIR BCE.

The content-addressable memory in our naive implementation of the TMT unit performs lookups in $O(1)$ time. However, in Table I we see that in terms of hardware resources the implementation does not scale well when the number of TMT rows r increases. Other implementation techniques can improve the resource scaling issue, but probably at the cost of increased lookup latency; this issue is dealt with in Section V-B.

B. Impact of the TMT Unit's Latency on the Processor Efficiency

We evaluate the impact of the TMT latency on the processor execution efficiency. The processor must cope with additional

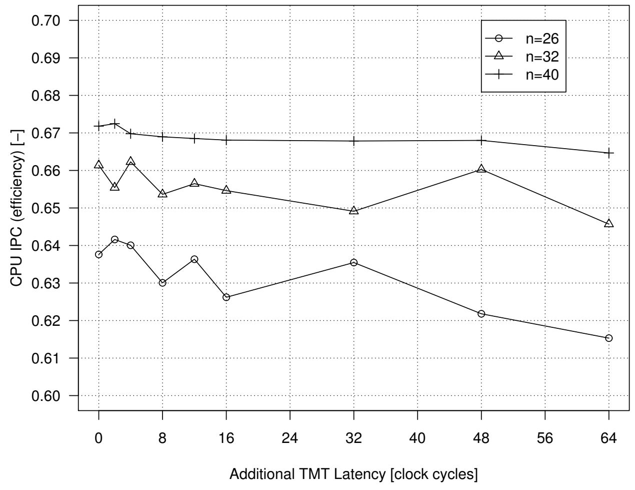


Figure 4. CPU IPC degradation if the TMT unit's latency hypothetically increases.

latencies as all family create events are routed through the TMT unit. The latency tolerance can be measured in terms of the CPU efficiency, or IPC (*Instructions Per Cycle*):

$$IPC = \frac{IC}{CC} \tag{2}$$

where IC is the instruction count, and CC is the total running time in clock cycles.

We ran 8 identical FIR computations in software to simulate higher workload and to average out transient effects of the dynamic thread scheduling. We limited the number of concurrently scheduled FIRs to at most 3 by the *blocksize* parameter so that we obtain a more realistic (less parallel) workload.

Baseline Efficiency Degradation: Table II shows the IPC degradation when the TMT unit is switched on. All *create events*, including those not belonging to any hardware accelerated family, are routed through the TMT unit, but no families of threads are actually executed in the accelerators (otherwise the instruction count IC would no longer be constant). We see that in the worst case (i.e. a short FIR computation: $n = 26$, $t = 24$) the IPC degradation is 0.025, but typically it is less than 0.01. The efficiency degradation in the worst case is caused by a higher ratio of the create events to compute operations in the processor.

TMT Latency Scaling: So far we have assumed the TMT unit's latency is only a few clock cycles. An implementation of the

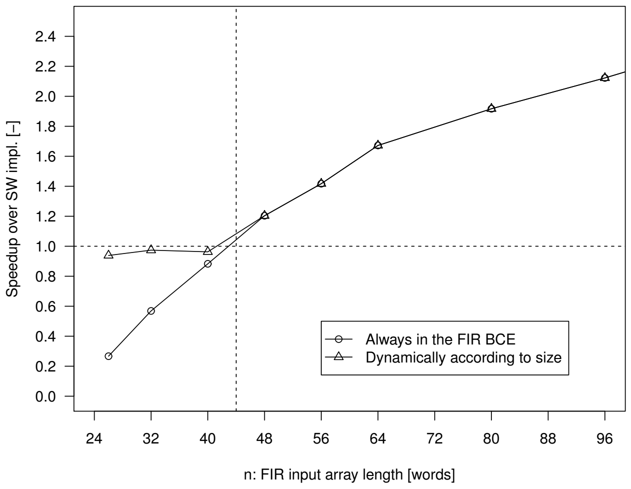


Figure 5. Speedups (including data transfers) over an all-software implementation for short input vectors n where accelerator overheads are most pronounced. Single FIR BCE, $t = 24$ taps, $24 \cdot (n - 23)$ MAC operations.

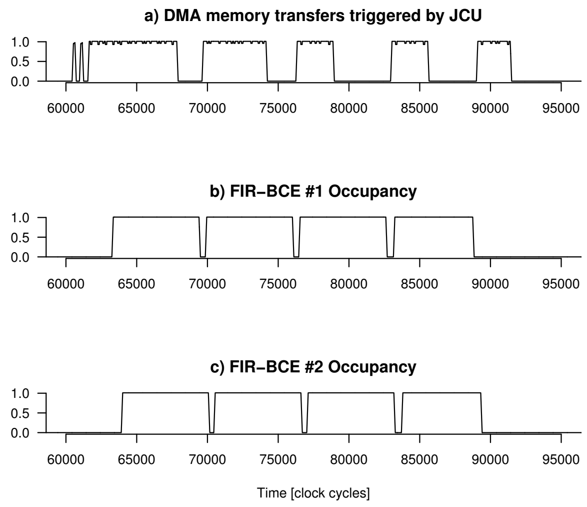


Figure 6. An example of concurrent control of two accelerators handled by the Job Control Unit. Each accelerator computes one family of threads (FIR with $n = 1024$). Each BCE has to be triggered four times (with the data transfers being scheduled in the ping-pong manner in background) as its local memories cannot hold the whole data at the same time.

TMT unit with a large number (tens to hundreds) of rows r would allow us to simultaneously map many classes of families of threads to hardware. (One example is to map kernels from the *BLAS*[6] package.) However, the clock cycle latency of such an implementation will be probably on the order of $O(\log r)$, or even $O(r)$ if sequential search is used.

To simulate and evaluate the situation we modified our TMT unit to impose an additional artificial non-pipelined latency during its processing of the create events that come from the processor. Figure 4 plots the efficiency IPC with respect to the additional latency. In the worst case (i.e. a short FIR computation: $n = 26$, $t = 24$) the IPC degrades from 0.640 to 0.615 (i.e. by 0.025) when the TMT latency increases by 64 cycles. This shows that the microthreaded processors tolerates the latencies introduced by our coupling scheme and validates the approach.

C. Filtering Short Families of Threads in the TMT Unit

The FIR BCE computes one MAC[7] per cycle while the software implementation requires at least 4 cycles for the same computation (2x LOAD, 1x MUL, 1x ADD). However, the main disadvantage of the accelerated hardware implementation is the need to transfer the I/O data to/from the main memory. In the experiments we considered the worst case scenario: at the beginning the data were live in the processor cache, they had to be flushed to the main memory, then downloaded into the BCE local buffer memory by the DMA engine in HWFAM, and at the end the results uploaded back to the main memory. Combined with other control overheads in the JCU the software implementation of a family can easily be faster.

Figure 5 shows a speedup of a *single* FIR computation with respect to an input vector length n. The number of the FIR taps is constant ($t = 24$). Recall that $n \geq t$ must hold and $(n-t+1)$ is the output vector length. The computation requires $t \cdot (n-t+1) = (nt - t^2 + t)$ MAC operations, and $(t+n+(n-t+1)) = (2n+1)$ words have to be transferred over the system bus (and evicted/flushed from the processor D-Cache beforehand, thus roughly doubling the actual number of clock cycles).

Indeed, the sample accelerated hardware FIR BCE reaches a speedup of 2x for $n = 96$ (i.e. 1752 MACs); however for $n < 44$ (i.e. less than 504 MACs) the software implementation is faster due to the data transfers – the ($\circ$) plot shows the speedup much less than 1 in that region. The microthreaded coupling scheme can amend the situation by filtering families of threads in the TMT unit according to their size so that short FIR families will be executed in software. This is shown by the ($\triangle$) plot which almost restores performance in the troublesome region. The remaining speedup degradation is due to the added latency of the *create events* introduced in the TMT unit.

D. Concurrent Accelerator Control

As the individual accelerator cores handle only the computation itself, the data transfers have to be managed by the Job Control Unit, including the ping-pong buffering required for computations involving large arrays. Figure 6 shows execution of two concurrent FIR computations in two FIR BCE cores managed by the JCU. As the FIR length was $n = 1024$, while the given implementation of the FIR BCE cores has the maximal vector length of only 256 elements, each accelerator has to be triggered four times, with data transfers taking place asynchronously. This is implemented in the JCU firmware by modelling the buffer memories as pools with low/high watermarks, and the DMA engine and BCE cores as general data pumps.

VI. Conclusion

The proposed microthreaded scheme for coupling custom accelerators to SVP processors was demonstrated in a sample FPGA

[6]BLAS = Basic Linear Algebra Subprograms
[7]MAC = Multiply-Accumulate

implementation. The benefit of the microthreaded coupling is that it handles concurrency in a uniform manner both in software and in hardware accelerators.

Selected parts of user computation, called families of threads, are implemented both in software to be executed in the processor, and as hardware configurations for a hardware accelerator. We have introduced a Thread Mapping Table (TMT) hardware unit that couples the software and hardware implementations, and modified the UTLEON3 processor to route all family creation events through it. Family creation requests that can be executed in hardware are transparently handed over to a Job Control Unit: a simple secondary controller that handles DMA data transfers between the main memory and local buffers, and commands the accelerator core(s) to execute a given function; at the same time the main processor continues executing other unrelated families of threads.

In the experimental section we have focused on specific features of the proposed design and evaluated the possible negative effect of increased latencies in the processor. The filtering of short families of threads in the TMT unit ensures that the custom hardware accelerators are not invoked if the expected overhead outweighs the speedup. On the other hand, it was shown that in the worst case the coupling scheme can degrade the processor efficiency by 0.025 in our case study, but typically less than 0.01.

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 [16].

References

[1] F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable instruction set processors from a hardware/software perspective," *Software Engineering, IEEE Transactions on*, vol. 28, no. 9, pp. 847 – 862, 2002.

[2] M. Danek, J. Kadlec, R. Bartosinski, and L. Kohout, "Increasing the level of abstraction in FPGA-based designs," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, 2008, pp. 5 –10.

[3] F. Thoma, M. Kuhnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. Muller-Glaser, and J. Becker, "MORPHEUS: Heterogeneous Reconfigurable Computing," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, pp. 409 – 414.

[4] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," 1997.

[5] J. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, 1997, pp. 12 –21.

[6] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte, "The MOLEN Polymorphic Processor," *Computers, IEEE Transactions on*, vol. 53, no. 11, pp. 1363 – 1375, 2004.

[7] P. G. Zaykov, G. K. Kuzmanov, and G. N. Gaydadjiev, "Reconfigurable multithreading architectures: A survey," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 263–274.

[8] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer, "Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 4 pp.

[9] H. K.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS '06. New York, NY, USA: ACM, 2006, pp. 259–264. [Online]. Available: http://doi.acm.org/10.1145/1176254.1176316

[10] C. Jesshope, "A model for the design and programming of multicores," *Advances in Parallel Computing*, vol. High Performance Computing and Grids in Action, no. 16, pp. 37–55, 2008.

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

[12] M. Danek, L. Kafka, L. Kohout, and J. Sykora, "Instruction set extensions for multi-threading in LEON3," in *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, 2010, pp. 237 –242.

[13] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin, "Sparcle: An evolutionary processor design for large-scale multiprocessors," *IEEE MICRO*, vol. 13, pp. 48–61, 1993.

[14] A. Mikschl and W. Damm, "MSparc: A Multithreaded Sparc," in *Euro-Par'96 Parallel Processings: Second International Euro-Par Conference, Vol II, LNCS 1124*. Springer Verlag, 1996, pp. 461–469.

[15] 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.

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