Design Methodology for Offloading Software Executions to FPGA

Tomasz Patyk, Perttu Salmela, Teemu Pitkänen, Pekka Jääskeläinen, and Jarmo Takala

Abstract Field programmable gate array (FPGA) is a flexible solution for offloading part of the computations from a processor. In particular, it can be used to accelerate an execution of a computationally heavy part of the software application, e.g., in DSP, where small kernels are repeated often. Since an application code for a processor is a software, a design methodology is needed to convert the code into a hardware implementation, applicable to the FPGA. In this paper, we propose a design method, which uses the Transport Triggered Architecture (TTA) processor template and the TTA-based Co-design Environment toolset to automate the design process. With software as a starting point, we generate a RTL implementation of an application-specific TTA processor together with the hardware/software interfaces required to offload computations from the system main processor. To exemplify how the integration of the customized TTA with a new platform could look like, we describe a process of developing required interfaces from a scratch. Finally, we present how to take advantage of the scalability of the TTA processor to target platform and application-specific requirements.

Keywords Application-specific integrated circuits · Hardware accelerator · Computer aided engineering · System-on-a-chip · Coprocessors · Field programmable gate arrays

This work has been supported by the Academy of Finland under research grant decision 128126.

Tampere University of Technology
Department of Computer Systems
P.O. Box 553
FI-33101 Tampere
Finland
E-mail: tomasz.patyk@tut.fi

1 Introduction

The growing complexity of software applications running on the portable devices like mobile phones, smart phones, PDAs etc., call for the increase in the processing power offered by their CPUs. Typically, a RISC processor employed as a general purpose processing unit does not provide enough computational resources and the use of a specialized hardware accelerator is inevitable. A DSP co-processor is a common solution to speed up multimedia applications. Nevertheless how powerful the DSP processor is, a dedicated hardware will do the same task faster, consume less power, and take smaller silicon area.

Reconfigurable hardware in form of field programmable gate array (FPGA) makes an excellent solution for increasing the performance of an embedded system, as part of the application code can be offloaded from the processor. The performance increase requires careful planning though. Quite often the overhead of such arrangements, e.g., cost of data transfers between a CPU and an FPGA may be higher than the performance gain. Also the clock frequency of the FPGA is often much lower than the CPU. Therefore, the inherent parallelism of the application needs to be exploited efficiently. Finally, the traditional development style for FPGA resembles hardware design process, which requires that the designer has expertise in hardware structures. Additionally, application code is often in a form of software code, hence, offloading requires the description to be converted to RTL structure. Therefore, there is a need for a design methodology converting software partition to a hardware structure. The methodology could be used, e.g., by software designers without a deep knowledge on the hardware implementations, as rapid way of offloading computations to an FPGA.

In this paper, we describe a design methodology for offloading computations from a CPU to a FPGA. The proposed method allows a part of an application code, described in the C language, to be executed on the application-tailored processor, implemented on the FPGA. The method supports full ANSI C language; targets with an operating system; exploits DMA transfers to minimize the overheads, and allows the user to scale-up/down the computational resources. Our experiments show that this method is scalable and can exploit the inherent parallelism of the application. In addition, the designer makes his efforts on the higher abstraction level, thus deep knowledge on the hardware design is not needed. The paper extends our previous work in [1] by providing details of the proposed design methodology.

The remaining part of the paper is organized as follows. Section 2 presents a brief survey of other available tools automating the offloading process. Section 3 sketches the offloading of computations, Section 4 details the implementation methodology for the described accelerator blocks, Section 5 describes the platform specific interfacing, Section 6 discusses results for two different TTA designs, and Section 7 concludes the paper.

2 Related Work

Traditionally the C language has been used to implement DSP algorithms and applications. The large amounts of legacy C code turns attention to design methods capable of converting functionality described in C language to a hardware structure, as easily as possible. A large number of tools, taking C program as an initial description, is already available on the market. In theory, such tools could be used for a FPGA based acceleration. However, many tools have serious limitations, e.g., only a subset of C is fully supported, which makes the C to hardware conversion process more complex and time consuming. Furthermore, many tools generate only the RTL description of an hardware accelerator without support for the system integration. The user has to manually design the scheduling and communication mechanisms between the accelerator and host processor, build the interface units, and provide device drivers.

Synfora PICO [2] generates processor arrays from C programs. However, it supports only a limited subset of C. It also requires manual setting of parameters affecting the scheduling of operations. CoWare Processor Designer [3] is a toolset for designing application-specific processors (ASIP) and it is not a generic tool for converting C to HDL descriptions. Target IP Designer [4, 5] is another similar tool. AutoESL [6] supports high and low level parallelism but it does not support the full ANSI C language. Impulse CoDeveloper [7] is targeted for an FPGA based acceleration but it assumes a computational model comprised of the sequential processes communicating with each other. Therefore, it suits well only if the application consists of the independent processes receiving and emitting data streams. In addition, it does not support full ANSI C language.

Binachip-FPGA [8] targets also FPGA acceleration. In contrast to other tools, the description of the system is given as a compiled binary for a supported processor architecture instead of a C language source code. Cascade [9] is another tool which uses ARM, PowerPC, or MicroBlaze binaries as the description of the desired functionality. These tools inputting binaries instead of source code are assumed to be targeted to cases where the source code of the program is not available. Otherwise it is hard to justify the lower level input format given that even the C language is a very low level sequential language from which producing a parallel implementation is already often very challenging, even if the described algorithm is inherently parallel.

Catapult-C [10] generates a fixed function implementation instead of a processor-based one. As a drawback, generating the hardware implementation requires lots of user attention. C2H is a tool only for the Altera FPGA devices. It requires direct access to a memory, shared with the master processor. The tool supports only a subset of C and its external connectivity is based on Altera's Avalon bus. Cynthesizer [11] is another tool for rapid hardware generation. However, it requires using SystemC as well. In general, extensive modifications are required to the original ANSI C code [12]. NISC [13] is a tool for generating no-instruction-set-computer architecture processors from

C. On the architectural level the basic idea of the NISC, the use of an extremely “bare bone” processor template, is similar to the TTA template used in this work. However, full ANSI C is not supported.

In the proposed method, we target for supporting the full ANSI C descriptions and allow user to trade of execution time against area according to given requirements. In addition, the proposed method supports offloading on targets with operating systems (OS).

3 Design Method for Offloading Computations

An FPGA in an embedded system gives a unique opportunity to system designers to offload some of the computation from the host processor, hence reducing the computational load on it. This hardware can serve as a hardware accelerator for some specific, e.g., DSP, algorithm that cannot be computed efficiently enough by the main unit. Another common case is to simply offload some computationally intensive tasks from the host processor in a multi-task system and let the processor execute other tasks while waiting for the results of the offloaded computation. Either way, the system designer is faced with the following design challenges:

- host processor utilization;
- hardware (HW) / software (SW) interface between the host processor and the offloaded unit; and
- co-design methodology to produce a HW accelerated implementation from the SW implementation.

When considering the host utilization, several issues need to be taken into account. Firstly, since the multi-tasking systems, governed by an operating system are of our primary interest, it is essential that the offloaded execution is non-blocking. This means that the host processor should be able to continue execution while the offloading hardware is doing its job. Quite often this means that the operating system schedules different tasks/processes to the processor until the execution can be resumed. Secondly, in some cases the FPGA system does not have a random access to the local memory of the processor where the operands of the computations are stored. This imposes the requirement of transferring data to and from the local memory of the FPGA device. Not only this takes time but also, if done actively by the host processor, it keeps the processor busy. A common way to avoid occupying the host processor for the data transfers is the use of Direct Memory Access (DMA) transfers. In the platforms supporting DMA, this method offloads the data transfers from the processor to a peripheral hardware unit. Thirdly, the FPGA circuit usually runs at a clock frequency several times lower than the one of the host processor. The actual acceleration expected from using the FPGA needs to be calculated keeping this in mind. Naturally, the gain arising from the fact that the host processor can perform other tasks meanwhile is preserved. These factors lead us to the following conclusion: in order to speed up application execution with an

FPGA accelerator, the speed of the accelerator hardware should compensate the additional data transfer penalties, the potentially lower clock frequency of the accelerator, and the overhead of task switching in the operating system. Preferably, the accelerator design technique should be scalable so it can be used to design accelerators that meet the required computational efficiency while staying within the silicon area limits of the platform.

The communication interface is specific to the used platform and hardware accelerator. If the accelerator is manually designed for the certain platform, the interface will be a direct map to the interface exposed by the platform. If the accelerator is generated with an automated approach, e.g., using a processor template, the need for an adapter interface is most certain. Should the DMA be exploited the interface needs to implement the means to enable this functionality. The interface is comprised of the hardware (HW) and software (SW) part. The HW interface establishes the signal connections between the system platform and the accelerator. The SW interface, in its basic form, allows data transfers to be performed, initiating the computations, and signaling the host processor about their completion.

For the design methodology, our approach is to design an application-specific processor for the task to be offloaded, and then use a retargetable C compiler to generate a binary code for the customized processor. We will also show how to create a HW/SW interface for an arbitrary platform. This interface requires non-recurring engineer work. Once created it can be reused on this particular platform with different application-customized accelerators. The HW and SW interfaces can be later distributed, e.g., in the form of reusable libraries.

4 Accelerator Implementation

In this work, the transport triggered architecture (TTA) [14] was used as a processor template for designing the accelerators. For design automation, TTA-based Codesign Environment (TCE) [15–17], that uses the TTA paradigm as a template for customizing application-specific processors, was used.

4.1 Processor Template

Transport Triggered Architectures (TTA) belong to a class of exposed data path VLIW architectures, i.e., the details of the data path transfers are exposed to the programmer. This enables various unique optimizations in code generation and the data path interconnection customization.

In contrast to traditional "operation triggered architectures" where operations are decoded to control signals that initiate operand transports, TTA instructions explicitly define and schedule the operand transports. The operation executions are side effects of the operand transports. The internal buses are used efficiently as the data transports on each bus can be controlled independently.

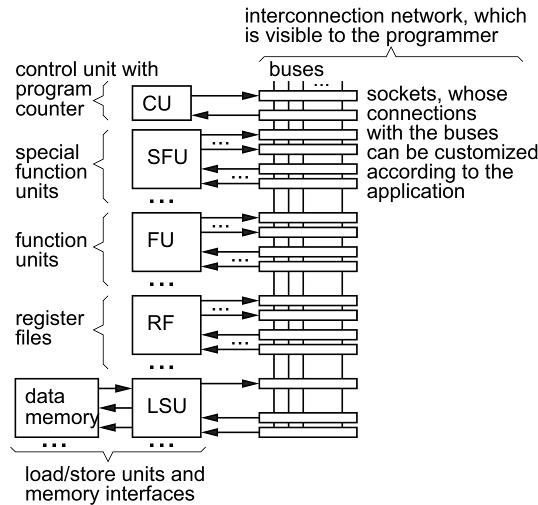

Fig. 1 TTA processors consist of the control unit (CU) and variable number of function units (FU), special FUs (SFU), register files (RF) and load/store units (LSU). Unused connections between the resources can be excluded from the interconnection network because of the data transport programming.

The modular structure of the TTA is illustrated in Fig. 1. Basic building blocks of TTA processors are function units (FU), register files (RF), a control unit, and an interconnection network between the data path resources. TTA processors are programmed by data transports between the computing resources and the programming paradigm reminds data flow programming. Each function unit contains one or more input ports. One of the input ports is a trigger port, which triggers the operation execution when the operand is moved to this port. This means that other operands have to be moved to corresponding ports on earlier or at the same instruction cycle as the move to trigger port. This requires careful scheduling of data transports. Operands can be passed directly from one function unit to another (*software bypassing*). Furthermore, the data can be often fully bypassed without the need for storing temporary results in a register file at all. In addition to reducing the number of needed general purpose registers to avoid spills, software bypassing lowers register file pressure, one of the biggest bottlenecks of the VLIW machines [18].

One of the main benefits of the TTA template is its flexibility; the architectures generated using the TTA template can be scaled to the requirements at hand. For instance, there are no limits on the number of parallel FUs or RFs. The FUs can have an arbitrary number of pipeline stages or an arbitrary delay. Furthermore, there is no limit on the number of input and output ports of FUs and the FUs can be connected to an external interface of the processor directly. The external interface is simply extended with the connected FU signals, which allows, e.g., using local memories freely. A second significant benefit is the simplicity and modularity of the processor, which alleviates verification and pre-synthesis cost estimations.

4.2 TTA-based Codesign Environment

The TTA-based Codesign Environment (TCE) [15–17] is a toolset that uses the TTA paradigm for developing application-specific instruction set processors. TCE offers a set of tools, which allow a designer to customize the processor architecture; compile high level language programs for the designed architectures; simulate the program execution; and evaluate the cost functions of execution cycles, area, and energy. The toolset includes both command line and graphical user interface tools for powerful scripting and comfortable usability.

TCE allows the designer to design processors completely manually or in a semi-automated fashion. In the first case, the designer uses a graphical tool to instantiate an architecture template and to populate it with resources. The library of predefined processor units include: register files, functional units, long immediate units etc. Additionally, the designer can add his own customized application-specific units. The graphical tool allows connecting processor resources with each other through the transport buses.

In the semi-automated design flow, the designer can automatically create an architecture based on the requirements of the application. Starting from an initial architecture provided by the designer, the *design space explorer* automatically adds and removes resources. Finally, the designer is given a database of architectures with associated information about the cycle counts required for executing the application.

The TCE design flow for FPGA circuits is illustrated in Fig. 2. The input is a high level language program. The first design space exploration loop is performed at the architecture level where the designed TTA is modified using graphical tools and evaluated using a retargetable compiler and a processor simulator. It should be noted that the “design space explorer” can be an automatic tool or the designer, depending on the desired design flow. The next phase is the hardware generation where a platform specific implementation of the architecture is produced. The implementation is then evaluated with platform vendor specific tools, which can return the design space exploration back to the architecture exploration in case the desired constraints (area, clock frequency, speed, power consumption) are not met.

The design variations are evaluated at architectural level by compiling programs for them and running architectural simulations. The C compiler is ANSI C compliant, hence, there are no restrictions on the C syntax. Once the designer is satisfied with the architecture the processor and proper program image can be generated. TCE tools generate the HDL files for the selected architecture and a bit image of the application. The processor architecture can be synthesized from the HDL files using third party tools.

In order to overcome the disadvantage of long instructions in VLIW designs the instruction compression can be used at this point. The binary image of the application is compressed and a corresponding decompressing block is added to the control unit of the target processor. For a more detailed description of the TCE FPGA design flow, the reader is referred to our previous paper [16].

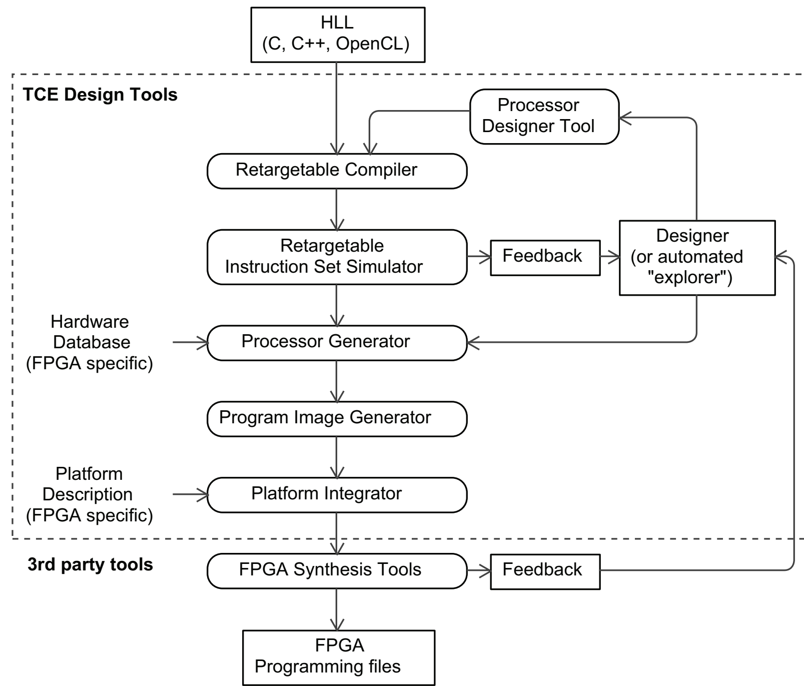

Fig. 2 TCE design flow for FPGA circuits.

4.3 Accelerator Design

The method for designing an accelerator on a FPGA for offloading computations from the host processor contains the following steps:

1. select a piece of code to be offloaded from the processor to the FPGA;
2. replace selected code by calls to the device driver to initiate operand transfers and execution on the FPGA;
3. customize a TTA processor for the selected code with the aid of the TCE-toolkit;
4. using TCE tools, generate an HDL description of the customized TTA processor with required interfaces from platform specific hardware databases and obtain the FPGA configuration with the commercial synthesis and place & route tools; and
5. generate machine code with the TCE retargetable compiler for the customized TTA.

At runtime the FPGA configuration is downloaded to the FPGA and the TTA binary program code is loaded to the FPGA memory. After the initializations the FPGA accelerator can be used under the software running on the host processor. The interfaces are to be loaded from platform-specific component libraries.

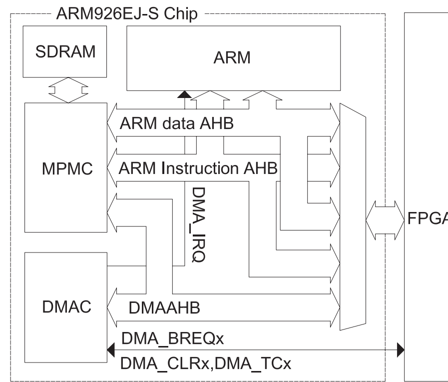

Fig. 3 Organization of the target platform. MPMC: Multi-port memory controller. DMAC: DMA controller.

5 Target-Specific Interfacing

The communication between the host processor and the application-specific TTA processor(s) configured on the FPGA is target dependent. Therefore, interfaces and protocols with device drivers are tailored for each target platform. However, once the tailoring has been done, the interfaces and protocols can be stored to libraries and reused for new applications.

5.1 Hardware Interface

Our example target platform was RealView Platform Baseboard for ARM926EJ-S, which contains the ARM processor and an FPGA chip. The simplified block diagram presenting the main components and their connections is shown in Fig. 3.

In this platform, all peripherals, which have a memory-mapped interface, communicate with the processor through the ARM specific AMBA AHB bus. Fig. 4 shows the basic connection of the slave peripherals to the tri-state AMBA AHB bus [19]. All AHB slave modules have their inputs permanently connected to the AHB signals. Outputs on the other hand are multiplexed. The Decoder component resolves addresses from the AHB address bus (HADDR) and activates the right component both by setting its HSEL signal high and multiplexing its output back to the AHB bus.

Since TTAs use the Harvard architecture, their interface is comprised of the separate busses to instruction and data memories. Additionally, our TTA included two control signals: input TTA_START and output TTA_COMPLETE. Those signals were used to start the computations and indicate that the results are ready. Once TTA_COMPLETE signal is asserted, the TTA is locked and does not perform any tasks. This prevents from possible data corruption and allows safe copying of results from the memory on the FPGA to a memory accessed by the host processor.

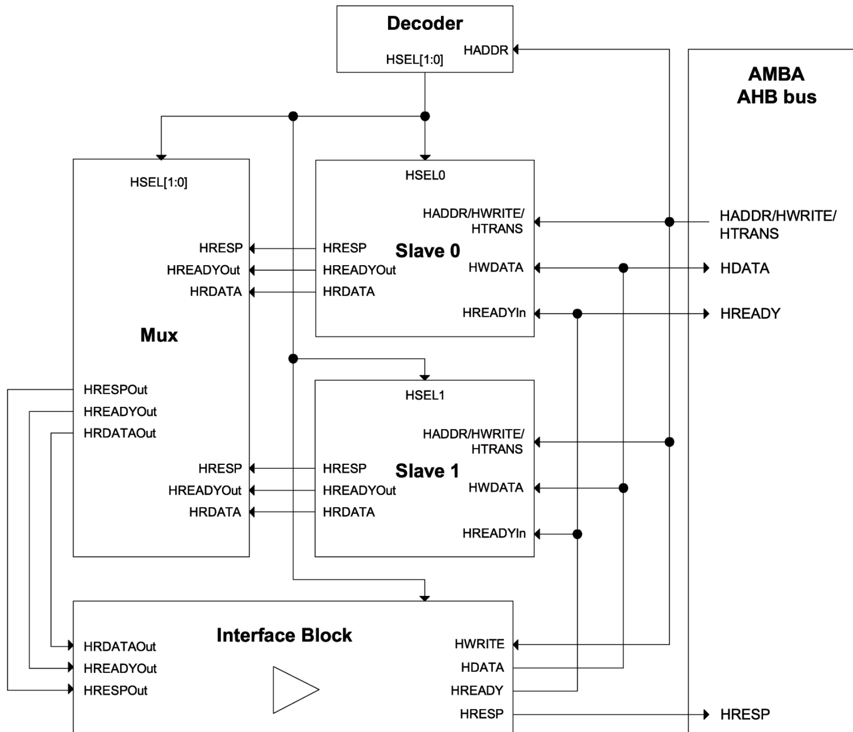

Fig. 4 Connection of AMBA AHB slaves [19].

The adapting interface between the target platform and TTA, presented in Fig. 5 is realized through three distinctive components instantiated on the FPGA: the data memory, the instruction memory and the DMA module (DMAM). Both memories are AMBA AHB slaves. The data memory is a dual-port RAM built from the on-chip memory cells on the FPGA. One port is connected to the TTA data memory interface, while the second port, which has an AHB interface, is connected to the AMBA bus.

The instruction memory is implemented in a similar way, with one exception. The ports are asymmetric in width. This is due to the very long instruction word of the TTA and, at the same time, the 32-bits width of the port connected to the AMBA bus. Because of this asymmetry, additional control logic is needed on the AMBA port to store and assemble several data words from the host processor, into a complete instruction word. This control logic is described with generic parameters, thus it can be reused easily by obtaining the details of binary code from the TCE compiler: memory size (the number of instructions to be stored); memory width (the instruction width); and word width (word with of data obtained from the host interface, in this case, the AHB uses 32-bit words).

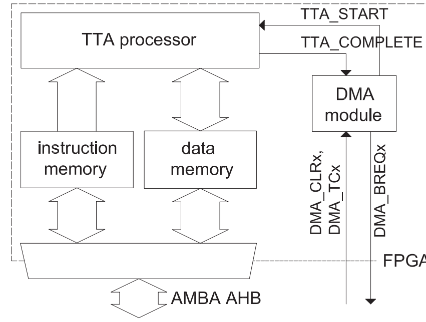

Fig. 5 Principal block diagram of application-specific processor in FPGA.

The actual data transfers on the FPGA are managed by the DMAM, which is also a simple finite state machine (FSM) that synchronizes DMA transfers with TTA processing and interleaves the accesses to the data memory. Typically, the following steps occur:

1. TTA is idle (locked) and does not access data memory, the DMAM enables DMA transfers;
2. DMA controller transfers data (divided into bursts) and the DMAM acknowledges consecutive bursts;
3. after the last burst the DMAM acknowledges transfer and unlocks the TTA which starts processing data in the memory;
4. once processing is done the TTA locks itself and informs the DMAM about the task completion; and
5. DMAM enables DMA transfers (pending or upcoming).

After the last step the DMA controller can setup the transfer back to the SDRAM. From the host processor point of view, offloading computations is nothing more than pushing data back and forth. The additional advantage comes from the fact that locking the TTA processor can result in significant power savings as the processor itself is neither polling nor waiting for an external interrupt.

5.2 Software Interface

The software interface is a platform specific driver. Our software platform was a Linux based OS, Maemo Scirocco [20], which is tailored for mobile systems. Therefore, we implemented the driver as Linux kernel module.

The host-slave communication is managed by the host processor through the DMA controller configured with the device driver. The driver is implemented as a kernel module, thus the driver can be dynamically loaded on runtime. The driver is implemented as a character device driver, which means that all the operations are performed on the file corresponding to the physical device. The list of system calls implemented by the DMA driver can be found in Table 1.

Table 1 System calls implemented by DMA controller Linux driver

System Call	*Implementation description*
`open`	Initiates driver specific structure. Opens a channel to communicate with the DMA controller.
`close`	Finishes an on-going transfer (if any) and clears private data.
`ioctl`	Sets transfer parameters, e.g., channel, number of bytes to be transferred, source and destination addresses etc. In the basic case, configuring a DMA transfer requires setting parameters in four registers in the DMA controller (DMACCxSrcAddr, DMACCxDestAddr, DMACCxControl, DMACCxConfiguration). More details can be found in DMA controller documentation [21].
`write`	Triggers transfer from SDRAM to FPGA. All necessary parameters need to be setup with the Ioctl beforehand.
`read`	Implements a blocking read operation. Triggers transfer from the FPGA to the SDRAM. This transfer might be blocked by the DMAM until TTA finishes processing. The transfer parameters need to be setup with the Ioctl beforehand.
`mmap`	Maps buffer from the kernel space to the user space. Mmap is required to make the same buffer visible in both spaces. It must be visible in the kernel space for the DMA controller and in the user space for the application. Data copying between kernel and user spaces is avoided by using the same buffer.

The developed driver supports both non-blocking and blocking data transfers. The driver implements also the DMA interrupt service, which is used to wakeup application during the blocking read. Since the DMA interrupt is enabled per transfer, it is important to enable it by the `ioctl` system call before blocking read is issued. Fig. 6 presents a typical use of the DMA controller driver system calls in the application program.

```
/* Open device */
fd0 = open(CHANNEL0, O_RDWR);
/* Allocate DMA buffer in kernel space */
ioctl(fd0, PL08X_IOC_ALLOC_SDRAM_BUFF, 8 * BUF_SIZE);

/* Map buffer from kernel to user space */
buffer = mmap(NULL, BUF_SIZE,
PROT_READ | PROT_WRITE, MAP_SHARED, fd0, 0);
/* Setup DMA controller registers */
ioctl(fd0, PL08X_IOC_SET_ALL, &dmac_c_params);
/* Enable DMA interrupt */
ioctl(fd0, PL08X_IOC_SET_DMA_IRQ, 1);

...

/* The write and read system calls
 * replace the call to offloaded function
 * in the original code. */
/* Transfer data from SDRAM to FPGA */
write(fd0, NULL, 0);
/* Blocking read until offloading is done */
read(fd0, NULL, 0);

...

/* Unmap buffer */
munmap(v->work[0], 8 * BUF_SIZE);
/* Close device */
close(fd0);
```

Fig. 6 Example of offloading code with blocking call.

5.3 Processor - Accelerator Interaction

Figure 7 presents the sequence diagram describing how the host processor operates with the accelerator during the program execution. Assuming that the FPGA has already been configured for the given application the interaction is carried in the following fashion. First, the application is started on the ARM processor. When the offloading should start, the host processor configures the DMA controller to perform a block transfer from the SDRAM to the TTA local memory in FPGA and starts the transfer. The host processor is now free for executing other tasks. After the DMA block transfer is completed, the TTA processor immediately starts processing the data. Once the TTA processor has completed the processing, it signals the end of the processing for the DMA controller such that the DMA transfer from the FPGA to the SDRAM could be initiated. When the transfer is finished, the DMA controller signals the host processor with the interrupt that offloading is completed and results are available. The interrupt service routine of the DMA device driver signals the operating system for context switch and the application continues its execution. On the consecutive offloading events the procedure is repeated.

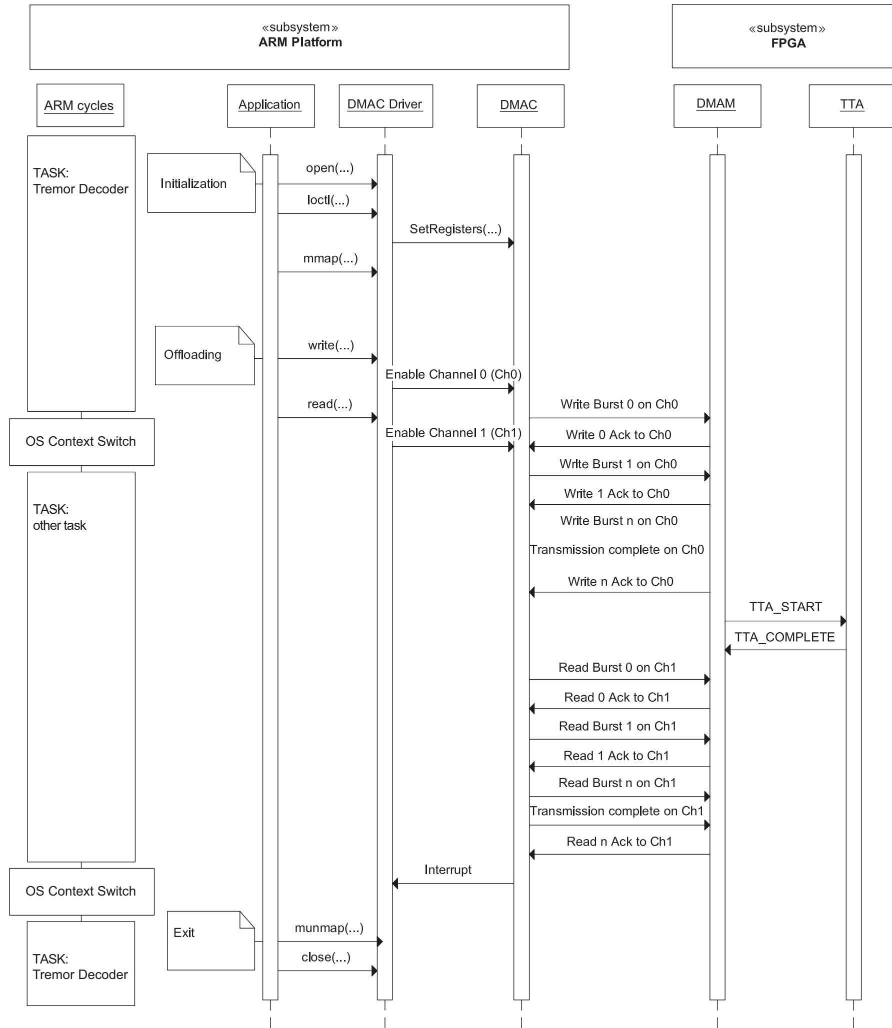

Fig. 7 Sequence diagram of a program execution with offloading.

6 Experiments

To proof the feasibility of the proposed methodology, we carried out experiments with the RealView Platform Baseboard, equipped with a Xilinx Virtex-II family FPGA. At heart of the board is ARM926EJ-S, the 32-bit RISC processor with a wide range of peripherals including the DMA controller (DMAC) and the memory management unit (MMU). The board contains also 128MB of the 32-bits wide SDRAM and 128MB of the 32-bits wide NOR flash memories.

The proposed design methodology was experimented by using the Tremor Ogg Vorbis audio decoder [22] as an example application. It is an open-source, fixed-point implementation of the standard, designed especially for platforms

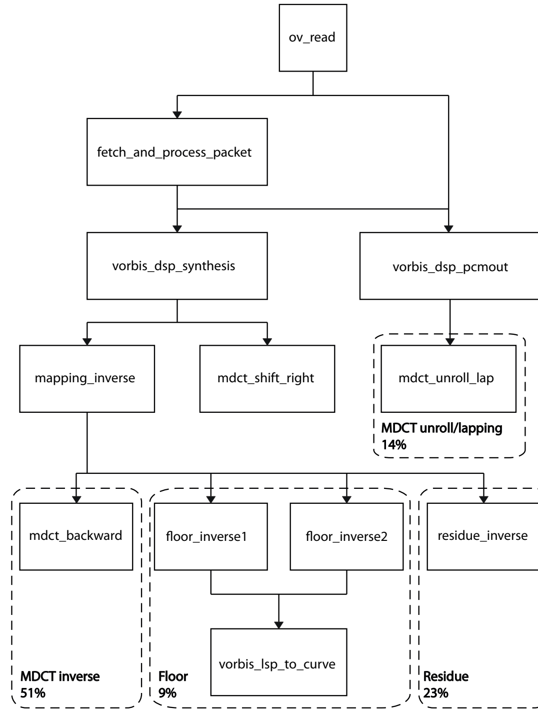

Fig. 8 Flow diagram of Tremor Ogg Vorbis audio decoder.

without floating-point arithmetics. Instead of compiling code directly on the board we decided to cross-compile it with the Scratchbox cross-compiling toolkit [23] run on the i686 Linux based host machine.

Finding a part of the application suitable for offloading is not trivial in a general case, especially with large programs. Fortunately in situations when the computational kernel cannot be easily identified, profiling tools, like *TCE's proxim* or *GNU's gprof*, can be used. The profiling provided information about the most complex functions in the Tremor Ogg Vorbis decoder. In Fig. 8, the flow diagram of the decoder along with the percentage of clock cycles used by the most significant parts of the application. Nearly 50% of computation time was used to compute the modified discrete cosine transform. As this function processes data in a consistent memory range, it was an obvious candidate for offloading. We built the customized TTA processor for the MDCT with the aid of TCE tools. To illustrate the scalability of the tools, we developed two processors. First, targeting to the short execution time, and second, aiming at the smaller area. We call them as *fastTTA* and *smallTTA*, respectively. The starting point was a so called *minimal architecture*, which contains just enough resources for the TCE compiler to compile any program for. The function, computing MDCT, was extracted from Tremor code and wrapped to a main function in a separate file. The TCE compiler supports ANSI C language so no other modifications were done to the original code. The code was compiled and

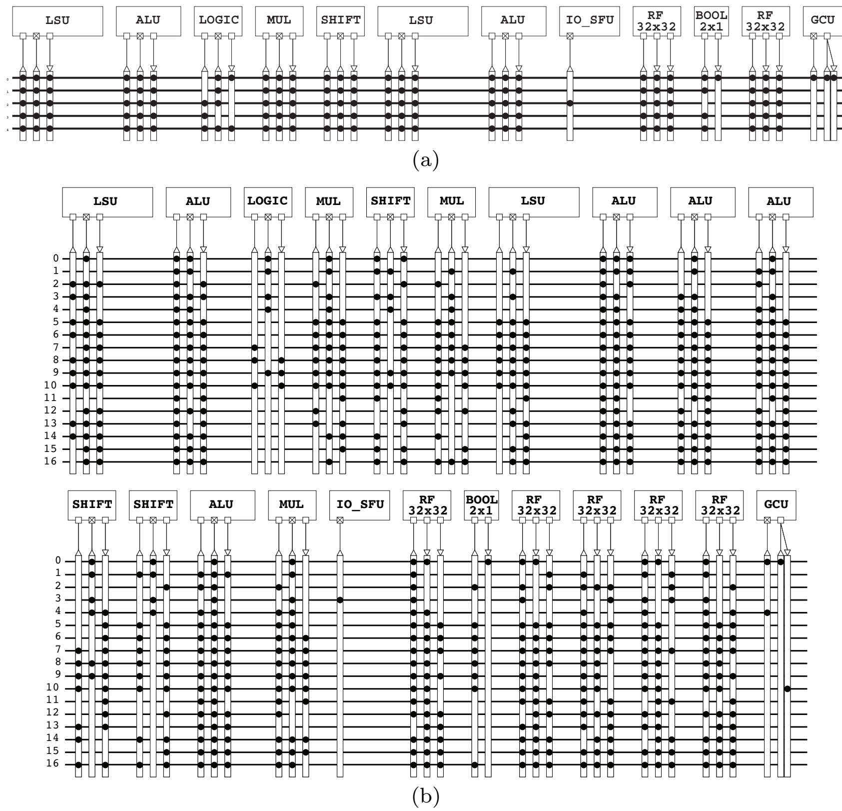

Fig. 9 Principal organization of the customized TTAs: (a) machine with limited resources, "smallTTA" and (b) higher performance machine, "fastTTA".

profiled in a cycle accurate simulator. The profiling tool shows the utilization of each function unit, hence, the often used FUs were duplicated to improve performance. The final configuration is given in Fig. 9(a).

The fastTTA, partially presented in Fig. 9(b), was obtained with the design space explorer tool from the TCE toolset. This tool automates the design process by adding resources iteratively until the cycle count cannot be reduced any more. Compared to smallTTA, fastTTA has following components in addition: two multipliers, three ALUs, two shifters, two register files, and 12 buses. The profiling, code modification and design of two application-specific TTA processors took approximately two days of work.

Two TTA machines were integrated with the rest of the hardware system from Fig. 5. Both designs were synthesized with the Xilinx ISE Design Suite 10.1. Table 2 presents some results taken from the synthesis and place & route reports. As we can see, the fastTTA takes almost three times more FPGA slices than the smallTTA due to the large number of FUs and interconnect buses. Also the difference in number of multipliers is significant. The fastTTa

uses nine embedded 18-bit multipliers, when the smallTTA has only three. The on-chip memory is also almost 50% larger when the fastTTA is used.

The difference in performance of TTAs represents T_{Comp} value, measured by the clock cycle accurate simulator from the TCE toolkit. The smallTTA takes 68315 cycles to execute the offloaded task, while the fastTTA executes the same routine in 50639 cycles.

The critical path of both designs, given in Table 2, is affected by two factors. Firstly, the synthesis was made for a relatively old FPGA architecture, namely Xilinx Virtex II. As example, clock frequency of $191MHz$ with a similar TTA processor was obtained when synthesized for a modern Xilinx Virtex-5 FPGA [16].

Secondly, no manual optimizations were used to optimize the critical path. Much higher clock frequencies could be obtained with manually optimized interconnect buses. That manual optimization can be easily applied with the help of graphical user interface from one of the tools in TCE. It is worth mentioned that this optimization process does not require a hardware design expertise from a designer. The longer critical path of fastTTA is due to the more complex interconnection bus. The complexity of the bus increases with the number of FUs and RFs in the design and the bus is often in the critical path of the FPGA implementations of TTAs.

Table 2 Characteristics of the offloading compared to software-only implementation.

	FastTTA	*SmallTTA*	*ARM*
FPGA slices	15412	4960	N/A
FPGA memory [kB]	86	54	N/A
FPGA Mul18 blocks	9	3	N/A
Max clock frequency [MHz]	35.90	36.02	210.00
Critical path [ns]	27.85	27.77	N/A
T_{Comp} [clock cycles]	50639	68315	682500
T_{Trans} [clock cycles]	9216	9216	N/A
T_{OS} [clock cycles]	˜1000	˜1000	N/A
$T_{Offload}$ [clock cycles]	60855	78531	682500
Offloaded code size [bytes]	21064	10986	7380

To measure the execution time of the application, as accurate as possible, we instantiated one additional component to the FPGA, cycle counter, which simply measures the number of FPGA clock cycles. The component has memory mapped registers, which allow start, stop, reset, and read of the measured clock cycles. The cycle counter is an AMBA AHB slave and can be accessed by the ARM processor exactly in the same way as any other memory mapped peripheral in the system.

The execution time of the accelerated function $T_{Offload}$ can be split into three distinct parts:

$$T_{Offload} = T_{Comp} + T_{Trans} + T_{OS} \tag{1}$$

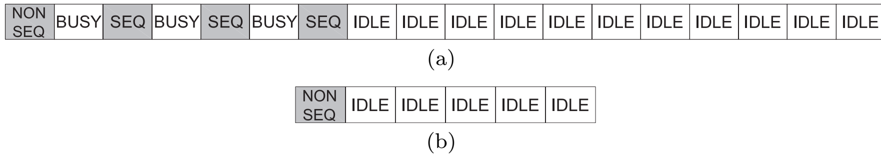

Fig. 10 AMBA HTRANS signal messages during a data transfer: (a) burst and (b) non-burst transfer. Each box corresponds to one clock cycle. Dark boxes indicate cycles when valid data is presented on the bus.

where T_{Comp} is the time used by TTA on the computations, T_{Trans} indicates the time of data transfers and T_{OS} reflects OS overhead of the master/slave communication. In our experiments, $T_{Offload}$ was measured with the cycle counter. The exact value of T_{Comp} can be calculated with the cycle accurate simulator from TCE tools. T_{Trans} can be computed based on the transfer protocol and the number of data elements to be transferred. Based on the previous the T_{OS} can be calculated according to (1).

Table 2 lists also the execution time results. The number of clock cycles the offloading takes is compared to the cycles that host processor needs to perform same computations. However, to correctly interpret these results we need to take into account that generally, the host processor runs at the higher frequency than the slave processor. In our case, the ratio equals 7. Keeping that in mind we obtain $1.6x$ and $1.24x$ speedup when offloading with the fastTTA and smallTTA, respectively. Additional gain comes from a fact that the host processor can perform other task while waiting for offloading to complete. We are running a multitask OS and other processes can be scheduled to run on the CPU during that time, as shown in Fig. 7.

The number of bytes the offloaded function takes after compiling is given in the last row of Tab. 2. As can be seen, the program for fastTTA is almost twice as big as the binary for the smallTTA. Conserving available memory on the FPGA for other purposes can be another reason to customize the accelerator exactly to the application requirements.

Finally, the reason for relatively low data transfer throughput is the transfer protocol used on the AMBA bus. Fig. 10 shows messages transferred from the master to the slave on the *HTRANS*, one of the AMBA signals. There are four distinct messages but only *NONSEQ* and *SEQ* indicate valid data on the bus. If we take a closer look at the messages during the burst transfer, shown in Fig. 10(a), we will see that 18 clock cycles are required to transfer four words of data. In other words, 4.5 cycles per word. In non-burst transfer, depicted by Fig. 10(b), one data word is send every 6 cycles. If we transfer 1024 words, which is the common case for the Tremor decoder, the transfer will take 4608 or 6144 clock cycles in burst and non-burst modes respectively. The bus we are using to transfer is not used for any other purpose, so it is safe to claim that calculated numbers hold in the general case. The burst mode can be set with the `DMACCxControl` registers of the DMA controller.

7 Conclusion

In this paper, we described a method for offloading computations from a host processor to an FPGA. The proposed approach supports platforms with an operating system and offloads both computation and data transfer between host and slave processors. The computations are implemented as a TTA processor, which is customized for the given application and exploits the inherent instruction level parallelism of the application. The interfaces and communication between the host processor and the slave TTA are target-specific but can be reused in the same target. The communication packages and interfaces are generic and allow any type of functionality to be offloaded from the host processor under this environment.

As a case study, we customized two TTAs for an audio decoding application, showing the scalability of the TCE toolset. The obtained results demonstrate that the difference in the targeted parameters is significant and the final product can be a trade-off based on the requirements. The design work is done with TCE tools on the high abstraction level, thus no hardware design expertise is needed. Finally, in our experiment, the results show that offloading speedup the application execution when compared to the software-only execution. However, the speedup depends on characteristics of the processor and FPGA fabric. Additional gain comes from a fact that the offloading is a non-blocking procedure. In a multitask operating system, other process can be scheduled to run on the CPU while the offloading is taking place.

References

1. T. Patyk, P. Salmela, T. Pitkänen, and J. Takala, "Design methodology for accelerating software executions with FPGA," in *Proc. IEEE Workshop Signal Process. Syst.*, Cupertino, CA, USA, Oct. 6–8 2010, pp. 46–51.
2. Synfora Inc., "Product brochure > PICO express FPGA," Mountain View, CA, USA, 4 p. [Online]. Available: http://www.synfora.com
3. A. Hoffman, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr, "A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language," *IEEE Trans. Comp.-Aided Design of Integr. Circ. Syst.*, vol. 20, no. 11, pp. 1338–1354, 2001.
4. J. V. Praet, D. Lanneer, W. Geurts, and G. Goossens, "Processor modeling and code selection for retargetable compilation," *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, no. 3, pp. 277–307, 2001.
5. Target Compiler Technologies, "IP designed | IP programmer," Leuven, Belgium, 2008, 4 p. [Online]. Available: http://www.retarget.com
6. J. Cong, "A new generation of C-based synthesis tool and domain-specific computing," in *Proc. IEEE Int. SoC Conf.*, vol. 6507, Newport Beach, CA, USA, Sept. 17–20 2008, pp. 386–386.
7. Impulse Accelerated Technologies Inc., "Accelerate C in FPGA," Kirkland, WA, USA, 2007, 2 p. [Online]. Available: http://www.impulsec.com
8. R. Goering, "Programmable logic: Startup moves binaries into FPGAs," *EE Times*, 2006.
9. CriticalBlue Ltd., "Cascade programmable application coprocessor generation," Pleasance Edinburgh, United Kingdom, 2007, 4 p. [Online]. Available: http://www.criticalblue.com

10. Mentor Graphics Corporation, “Catapult C synthesis datasheet,” Wilsonville, OR, USA, 2010, 4 p. [Online]. Available: http://www.mentor.com/catapult
11. Forte Design Systems, “Cynthesizer^{TM} the most productive path to silicon,” San Jose, CA, USA, 2008, 2 p. [Online]. Available: http://www.forteds.com
12. “ESNUG ELSE 06 Item 7 Subject: Mentor Catapult C,” 2006. [Online]. Available: http://www.deepchip.com/items/else06-07.html
13. M. Reshadi and D. Gajski, “A cycle-accurate compilation algorithm for custom pipelined datapaths,” in *Proc. IEEE/ACM/IFIP Int. Conf. HW/SW Codesign System Synthesis*, New York, NY, USA, Sept. 18–21 2005, pp. 21–26.
14. H. Corporaal, “Design of transport triggered architectures,” in *Proc. 4th Great Lakes Symp. Design Autom. High Perf. VLSI Syst.*, Notre Dame, IN, USA, Mar. 4–5 1994, pp. 130–135.
15. P. Jääskeläinen, V. Guzma, A. Clio, T. Pitkänen, and J. Takala, “Codesign toolset for application-specific instruction-set processors,” in *Proc. SPIE Multimedia Mobile Devices*, vol. 6507, San Jose, CA, USA, Jan. 29–30 2007, pp. 05 070X–1–10.
16. O. Esko, P. Jääskeläinen, P. Huerta, C. S. de La Lama, J. Takala, and J. I. Martinez, “Customized exposed datapath soft-core design flow with compiler support,” in *Proc. Int. Conf. Field Programmable Logic and Applications*, Milano, Italy, Aug. 31 – Sept. 2 2010, pp. 217–222.
17. “TCE: TTA codesign environment.” [Online]. Available: http://www.tcs.cs.tut.fi
18. H. Corporaal, “TTAs: missing the ILP complexity wall,” *J. Syst. Architecture*, vol. 45, no. 12-13, pp. 949–973, 1999.
19. “Implementing AHB Peripherals in Logic Tiles. Application Note 119.” [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0119e/index.html
20. “Maemo by Nokia.” [Online]. Available: http://maemo.org/
21. “AMBA Open Specifications.” [Online]. Available: http://www.arm.com/products/system-ip/amba/amba-open-specifications.php
22. “Tremor by the Xiph.Org Foundation,” 2006. [Online]. Available: http://wiki.xiph.org/index.php/Tremor
23. “Scratchbox cross-compilation toolkit project.” [Online]. Available: http://www.scratchbox.org/