T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE
FOR EMBEDDED SYSTEMS

Project Number 288008

D 2.7 Evaluation report on interrupt virtualization

Version 1.0
3 September 2014
Final

Public Distribution

Eindhoven University of Technology

Project Partners: **AbsInt Angewandte Informatik**, **Eindhoven University of Technology**, **GMVIS Skysoft**, **Intecs**, **Technical University of Denmark**, **The Open Group**, **University of York**, **Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft João Baptista Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: joao.baptista@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First draft for review	21 August 2014
0.2	Updated after review	26 August 2014
1.0	Final version after QA	3 September 2014

Executive Summary

This document describes the deliverable *D 2.7 Evaluation report on interrupt virtualization* of Work Package 2 of the T-CREST project, due 36 months after project start as stated in the Description of Work. This document presents the evaluation of virtualized interrupts.

Deliverable D2.2 *Concepts for Interrupt Virtualisation* introduced the concepts for interrupt virtualisation, which were further developed into a design and implementation in Deliverable 2.4 *Design and Implementation of Interrupt Virtualisation Hardware and APIs*. The Interrupt Control and Timing Manager (ICTM) is the hardware accelerator of the Patmos processor [3] that implements required functionality in the T-CREST platform. This document describes how the ICTM coprocessor was integrated with the Patmos processor, from a hardware and software perspective. We then evaluate the hardware architecture defined in the project, and draw conclusions regarding alternative hardware approaches with different advantages and disadvantages. Finally we present recommendations and conclude.

1 Integration Overview

The ICTM coprocessor has been integrated in the T-CREST platform. The ICTM is a coprocessor of the Patmos processor since that is the hardware block with which is interacts most closely. Every Patmos processor has a single ICTM coprocessor. Figure 1 gives a high-level integration overview.

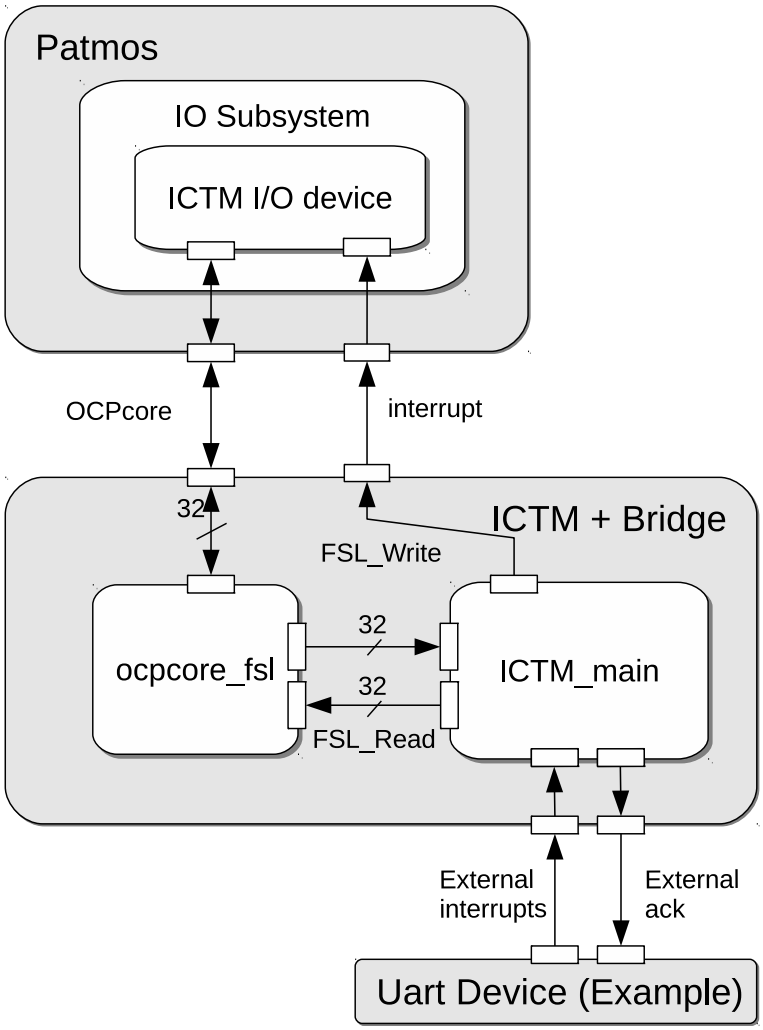

Figure 1: The Patmos processor, the ICTM coprocessor, and the ICTM coprocessor OCP-to-FSL bridge submodule.

For a modular design the ICTM coprocessor is a coprocessor that can be programmed and interrogated by the Patmos processor as a memory-mapped I/O device. The Patmos Handbook [3] defines how I/O devices are integrated in the Patmos subsystem, in Section 3.2 on I/O Devices. The Patmos processor uses the Open Core Protocol (OCP) for its hardware ports.

The ICTM coprocessor also requires interaction with the Patmos processor on the interrupt pins, since it is located between any devices that generate interrupts, such as a UART, and the Patmos processor. The Patmos Handbook [3] defines how interrupts are handled by the processor in Section 2.7 Exceptions: Interrupts, Faults and Traps. Form a hardware perspective the ICTM coprocessor is transparent

to the Patmos processor as well as the devices that generate interrupts. The ICTM coprocessor is also transparent from a software perspective for the devices. The Patmos processor is of course aware of the ICTM coprocessor from a software viewpoint, since it has to interact with the ICTM coprocessor using a software API.

2 Hardware Integration

To expose an additional OCP port and interrupt pin port on the Patmos processor, an extra ICTM coprocessor device was added to the I/O subsystem. This ICTM coprocessor device is accessible in the memory space of the processor at address 0xF0000A00 and 0xF0000A04. Hence all OCP communication with the ICTM coprocessor takes place through Memory Mapped I/O (MMIO). This is a convenient way to interact with custom hardware such as a coprocessor.

Additionally, the ICTM coprocessor device exposes an interrupt pin to the system outside of the Patmos processor. For non-virtualised interrupts, this interrupt pin is connected directly to the final device, such as a UART. Since the ICTM coprocessor virtualised interrupts, it is placed between the final device, such as a UART, and the Patmos processor, see Figure 1. Thus, the interrupt pin is connected to the interrupt output of the ICTM coprocessor. Acknowledgment of this interrupt happens by means of an MMIO transaction on the OCP port.

The ICTM coprocessor natively has an streaming read and write interface, called Fast Simplex Link (FSL). The FSL offers an independent read and write interface, instead of single read-write interface. To allow communication with the Patmos processor, an OCP to FSL bridge was developed. Note that this bridge is limited to the subset of OCP defined as OCPio in the Patmos handbook [3]. The ICTM coprocessor uses the FSL interface with the (simple) conversion block because in this way the ICTM coprocessor can be easily ported to other processors that do not use the OCP interface but other interfaces. For example, the ICTM coprocessor works directly with the Microblaze processor using the FSL interface, or using a wrapper, it could also work with the Microblaze processor interface using the Peripheral Local Bus (PLB), or using a wrapper it could also work with ARM processors.

Read and write transactions on the Patmos processor that have an address in the range that is mapped for the ICTM coprocessor are translated into streaming instructions. These instructions are internally interpreted and executed, see Figure 2.

The ICTM coprocessor is itself a simple processor that receives instructions by programming its memory-mapped registers. The instruction decode block then decodes and executes these instructions.

The ICTM coprocessor has an external interrupt input port that is connected to devices, such as a UART. It receives interrupt signals over this input. These interrupts are acknowledged, i.e. a signal is sent back to indicate reception of the interrupt, over the acknowledgement port.

This architecture makes the ICTM coprocessor relatively portable.

Internally, the ICTM coprocessor consists of a 64-bit timer that serves as the wall-clock, i.e. a timer that keeps ticking independently of the behaviour of the system. It also has two programmable counters (one for the system and one for application software), a clock-management module based on clock sub-sampling. The timer as well as internal counters are 64 bits wide to ensure that they do

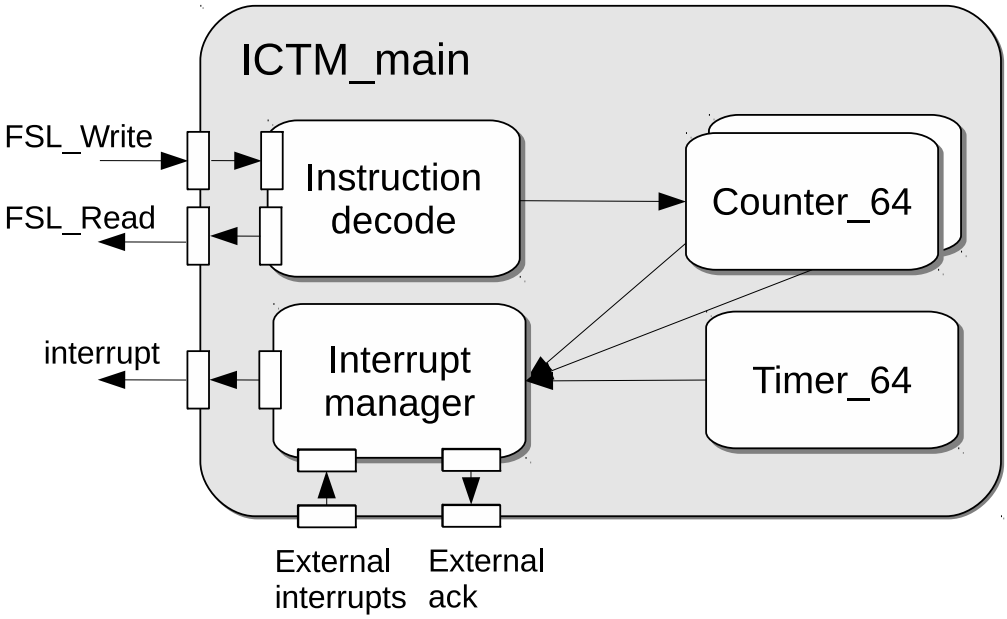

Figure 2: ICTM coprocessor internal structure.

not wrap around during the life time of the system: at 100 MHz (the speed of the Patmos processor) 64 bits allows counting up to 5850 years. 32 bit counters would wrap around after only 42 seconds, which complicates the software that handles timers, deadlines, etc. The ICTM coprocessor also contains an interrupt manager that merges / masks the different sources of interrupts into a single interrupt vector. The ICTM coprocessor is programmed through the FSL write port, and timer values and status registers can be read through the FSL read port. These are accessible through the single OCP port using write and read instructions on the Patmos processor respectively.

3 Hardware Evaluation

The architecture of the ICTM coprocessor hardware and software is versatile, since it is used in two projects with different processors. In the T-CREST project it is used with the Patmos processor, whereas in the Flextiles project (FP7-ICT1 288248) it is used with a Microblaze processor. (Of course, the ICTM is a deliverable and reported as output only in the T-CREST project.)

The generation of a Patmos-based platform with the required ports to allow the insertion of the ICTM coprocessor is automated through the regular T-CREST design flow (also described in the Patmos Handbook [3]). The T-CREST platform is prototyped on a Xilinx ML605 FPGA. The platform description for the ML605 instance with on-chip memory was extended for the purpose of this integration, such that the ICTM coprocessor I/O device is inserted in the generated Verilog code of Patmos. See Appendix B for details. A final manual step is then required to add the ICTM coprocessor bridge and ICTM coprocessor_main VHDL files to the project, after which the entire platform can be synthesized.

The ICTM coprocessor has been evaluated in terms of hardware complexity and cost. In particular:

Table 1: Hardware costs per sub-block, after synthesis

Component	ocpcore_fsl	ictm_main	counter_64_0	counter_64_1	interrupt mngr	timer_64	Total
D-type flip-flop	104	253	706	706	16	196	1981
64-bit adder		2	6	6		2	16
16-bit adder		1					1
32-bit adder			2	2			4
8-bit comparator		1					1
Multiplexer		872	60	60	18	6	1016
64-bit comparator			4	4		1	9
RAM 16x3-bit		1					1
RAM 8x2-bit	1						1

- The ICTM coprocessor consists of 1,777 lines of VHDL, and is separated in 5 sub-modules.
- The synthesis tool reports the maximum frequency to be 185 MHz, which is as fast as the Patmos processor. The ICTM coprocessor therefore does not impede the Patmos processor.
- Table 1 shows the approximate hardware cost per sub-block in the ICTM coprocessor after the initial synthesis step. Note that the counters are pipelined, such that the execution of arithmetic operations on them (e.g. updating with a new value, adding relative offsets, etc) are buffered. The selection of the next output value of the counter happens in a separate pipeline stage. Hence there are more flip-flops in the design than one would expect at first sight (64), but we require this extra hardware to maintain a reasonable clock frequency.
- Table 2 shows the approximate hardware cost after global optimization across all ICTM coprocessor components.
- Table 3 shows the mapping to FPGA primitives. The ICTM coprocessor uses 1746 slice registers, and 4311 slice LUTs. A single Patmos uses 3702 slice registers and 5008 slice LUTs. The ICTM coprocessor therefore occupies 47.2% slice registers and 86.1% slice LUTs, compared to a Patmos processor.

Besides functional correctness, the most important conclusion of the hardware evaluation is that the register and LUT cost of an ICTM coprocessor is very significant compared to the Patmos processor. We return to this point in a later section.

Table 2: Hardware costs per ICTM coprocessor, after advanced HDL Synthesis

Type	Amount
# RAMs	1
16x3-bit single-port distributed Read Only RAM	1
# Adders/Subtractors	21
16-bit subtractor	1
32-bit subtractor	4
64-bit adder	12
64-bit subtractor	4
# Registers	1872
Flip-Flops	1872
# Comparators	10
64-bit comparator equal	3
64-bit comparator lessequal	6
8-bit comparator lessequal	1
# Multiplexers	1403
1-bit 2-to-1 multiplexer	1161
1-bit 4-to-1 multiplexer	44
2-bit 2-to-1 multiplexer	1
32-bit 2-to-1 multiplexer	38
4-bit 2-to-1 multiplexer	10
6-bit 2-to-1 multiplexer	4
64-bit 2-to-1 multiplexer	72
8-bit 2-to-1 multiplexer	73
# FSMs	1
# Xors	1
8-bit xor2	1

Table 3: Hardware costs per ICTM coprocessor, FPGA primitives

Type	Amount
# BELS	6527
# GND	1
# INV	114
# LUT1	318
# LUT2	634
# LUT3	409
# LUT4	699
# LUT5	758
# LUT6	1379
# MUXCY	1206
# MUXF7	33
# VCC	1
# XORCY	975
# FlipFlops/Latches	1746
# FD	1
# FDR	133
# FDRE	1175
# FDS	17
# FDSE	420
# Clock Buffers	1
# BUFGP	1
# IO Buffers	84
# IBUF	42
# OBUF	42

4 Software Integration and Evaluation

As explained before, to expose an additional OCP port and interrupt pin port on the Patmos processor, an extra device was added to the I/O subsystem. This device is accessible in the memory space of the processor at address 0xF0000A00 and 0xF0000A04. Hence all OCP communication with the ICTM coprocessor takes place through Memory Mapped I/O (MMIO). In essence, a program on the Patmos processor can use write instructions to update programmable registers in the ICTM coprocessor, and use read instructions to retrieve the current state of the ICTM coprocessor. This is a convenient way to interact with custom hardware such as a coprocessor.

Writing and reading memory-mapped registers in a coprocessor is relatively error-prone for manual programming, since it required an intricate knowledge of the registers and bits of the coprocessor. We alleviate this problem by offering a higher-level software API in the C language. As an example, the handling of 64 bit counters over a 32 bit OCP interface with a 32-bit processor can lead to consistency problems, unless properly managed. The ICTM coprocessor hardware together with the software API ensure that is performed correctly. In particular, the internal counters that are programmable or readable are duplicated such that successive write or read instructions from the Patmos processor are first gathered in a single 64-bit register. Following this, a single 64-bit atomic update (either write/update or read/sample) is performed. Furthermore, some of the operations that the API performs are non-interruptible, i.e. they contain critical sections. Within the API, we therefore disable and enable the interrupts functionality of the Patmos processor at the appropriate times.

The ICTM coprocessor software API was defined in D2.4 *Design and Implementation of Interrupt Virtualisation Hardware and APIs*, Section 5.2. An improved updated version is included in Appendix A.

The ICTM coprocessor software API defined in D2.4 Section 5.2 was compiled successfully with the Patmos-llvm compiler. The size of the compiled driver is 7.8 kilobyte, which is relatively small.

The software API of the ICTM coprocessor is portable. It is used in two projects with different processors. In the T-CREST project it is used with the Patmos processor, whereas in the Flextiles project (FP7-ICT1 288248) it is used with a Microblaze processor. (The software API for the ICTM coprocessor is only delivered and report in T-CREST.)

After synthesis of the Patmos processor, each with its ICTM coprocessor, the generated bitfile is programmed onto an FPGA for additional function testing. The available bootloader infrastructure was used to load a test application into the platform. With this we verified that the ICTM coprocessor successfully comes out of reset, and is responsive to read/write transactions. By using the ICTM coprocessor software API, we further verified that the ICTM coprocessor functions as intended when it comes to programming it, i.e. configuring timers and interrupts works, and we can successfully extract status information from it. Finally, we verified that interrupts are generated by the ICTM coprocessor are indeed captured in the processor.

The ICTM coprocessor is part of a large system, and to illustrate its behaviour we include a waveform of some I/O signals in a Modelsim simulation. It is not possible to obtain these traces from the FPGA. The waveform is shown in Figure 3, which displays a subset of the signals that are passed between the ICTM coprocessor and Patmos through the bridge. The particular traffic in the picture is generated by a modified version of the bootloader as opposed to the test application, for debug-

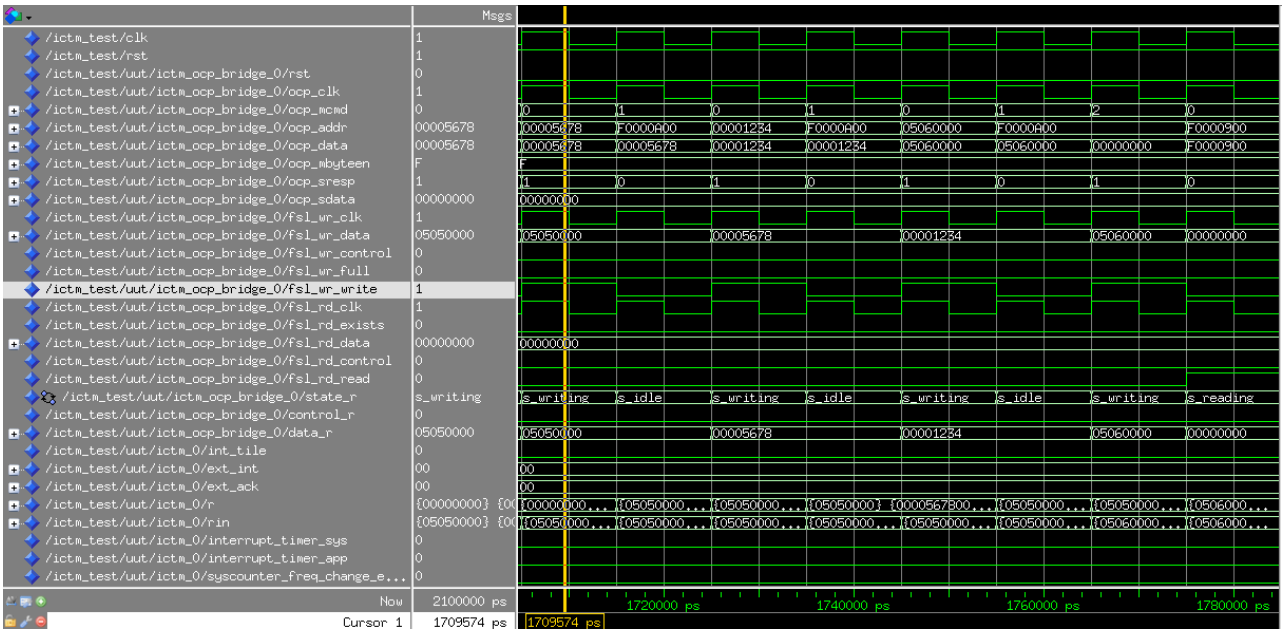

Figure 3: ICTM coprocessor communicating with Patmos through the OCP Bridge in a Modelsim simulation.

ging purposes. (Using the test application would require a port of the bootloader mechanism to the Modelsim environment, which is non-trivial).

A sample output of the Patmos processor extended with the ICTM coprocessor while running on the FPGA is shown in Listing 1.

```
(...)
----------------------------------------------------------------------
INFO:iMPACT:501 - '2': Added Device xc6vlx240t successfully.
----------------------------------------------------------------------
----------------------------------------------------------------------
>Maximum TCK operating frequency for this device chain: 16700000.
Validating chain...
Boundary-scan chain validated successfully.
2: Device Temperature: Current Reading: 33.64 C, Min. Reading: 31.67 C, Max.
Reading: 36.59 C
2: VCCINT Supply: Current Reading: 1.011 V, Min. Reading: 1.005 V, Max.
Reading: 1.014 V
2: VCCAUX Supply: Current Reading: 2.499 V, Min. Reading: 2.493 V, Max.
Reading: 2.505 V
'2': Programming device...
 Match_cycle = NoWait.
Match cycle: NoWait
 LCK_cycle = NoWait.
LCK cycle: NoWait
done.
INFO:iMPACT:2219 - Status register values:
INFO:iMPACT - 0011 1111 0101 1110 0000 1011 1100 0000
INFO:iMPACT:579 - '2': Completed downloading bit file to device.
INFO:iMPACT:188 - '2': Programming completed successfully.
 Match_cycle = NoWait.
Match cycle: NoWait
 LCK_cycle = NoWait.
LCK cycle: NoWait
```

```
INFO:iMPACT - '2': Checking done pin....done.
'2': Programmed successfully.
Elapsed time = 13 sec.
>------------------------------------------------------------------------
------------------------------------------------------------------------
Hello, World from an ictm-enabled patmos!
Enabled interrupts.
hw_ictm_systimer_set done 0
hw_ictm_systimer_enable_int done 0
hw_ictm_get_error_reg: 0x00000000 0
0: hw_ictm_systimer_get: 0x00000000 0x0008aaf6
1: hw_ictm_systimer_get: 0x00000000 0x000cc6f7
2: hw_ictm_systimer_get: 0x00000000 0x0010e2f3
3: hw_ictm_systimer_get: 0x00000000 0x0014fef1
4: hw_ictm_systimer_get: 0x00000000 0x00191af2
5: hw_ictm_systimer_get: 0x00000000 0x001d36f0
6: hw_ictm_systimer_get: 0x00000000 0x00215298
7: hw_ictm_systimer_get: 0x00000000 0x00 (...)
```

Listing 1: Sample of UART output for ICTM test run on the FPGA.

5 ICTM Coprocessor Architecture Evaluation

The major design rationale for the ICTM coprocessor is to decouple the interrupt virtualisation functionality from the processor. While this is a good idea architecturally in general, this turned out to be not optimal in T-CREST for the following reasons.

5.1 Disadvantages of Decoupling Patmos Processor and ICTM coprocessor

Hardware Area Cost

The hardware cost of the ICTM coprocessor, which can be up to 81% of the Patmos processor, is high. (The Patmos processor is relatively small, which is a good thing, but still the cost of the ICTM is high.) This resulted mostly from the fact that the counters in the ICTM coprocessor are 64 bits (to avoid wrapping around), and also that a number of the counters have to duplicated for pipelining and safe sampling, as described above.

Architecture Function duplication

Some functionality is required in both processor and ICTM coprocessor, leading to hardware duplication. This includes the expensive cycle counters and timers, and interrupt vectors.

Functional decoupling and complexity

The Patmos processor and the ICTM coprocessor are both programmable processors (the ICTM coprocessor is of course much simpler) with their own execution threads. As a result, they have to be

synchronised safely, which is currently done through the OCPCORE_FSL bridge and a corresponding software API. The decoupling hardware and software introduces implementation complexity. Since a high-level ICTM coprocessor software API is used, the complexity for the programmer is not affected.

5.2 Advantages of Decoupling Patmos Processor and ICTM coprocessor

Reasons to keep the processor and the ICTM coprocessor functionality and implementation separate from a processor include:

Closed vs. Open Processor

The T-CREST project has a bespoke hardware and software architecture to reduce the worst-case execution time of applications. The Patmos processor architecture and implementation are therefore completely under control of the project. The Patmos processor is in that sense an "open core." This allows interrupt virtualisation functionality to be integrated in the Patmos processor.

However, when the processor is a "closed core," such as a MicroBlaze processor or ARM processor, which means that it cannot be modified, the ICTM coprocessor remains the only valid proposition since its functionality cannot be added otherwise. We have demonstrated that the ICTM coprocessor can be integrated with a closed core Microblaze processor elsewhere.

Hardware vs. Software Complexity

Some of the ICTM coprocessor functions can be implemented in software only, as shown in [2, 1]. However, this is quite complex. It is thus hard to analyse and maintain [1] and dependent on the particular architecture and capabilities of the processor. This solution is therefore less portable than the ICTM coprocessor hardware with software API. In fact, some processors do not allow implementation of ICTM functionality at all in software [2].

Instead, the ICTM coprocessor hardware can remain the same for different processors, except for small modifications (or omission) of the OCPCORE_FSL protocol conversion block, and minor changes to the software API (i.e. how memory-mapped I/O is managed).

Power Management (Optional)

This point is not relevant (yet) for the T-CREST platform, since power management is not addressed in the project. However, if in the future this functionality is required, it would be relevant.

If real-time composable power management is required then it is suggested to keep the ICTM coprocessor functionality apart from the processor in a separate ICTM coprocessor. The reason is that clock generation is a separate module from logic, such as processor. An ICTM coprocessor would be integrated with the clock generation unit, since the keeping of time (i.e. the wall-clock counter) must be performed on an unscaled clock. The ICTM coprocessor and the processor are thus separated by

a clock domain crossing. The ICTM coprocessor is easily decoupled from a processor with a clock domain crossing in the OCPCORE_FSL module.

6 ICTM Coprocessor Architecture Recommendations

For the reasons listed in the previous section, it is proposed that future developments of the T-CREST platform functionally couple or merge the ICTM coprocessor functionality into the Patmos processor. For T-CREST this will lead to a simpler and cheaper overall design. In particular, the following are suggested:

Sleep Mode and ICTM Coprocessor Common Functionality

The Patmos processor has during the course of the project been extended with a Sleep Mode ([3, Section 2.7.6 Sleep Mode]). This Sleep Mode has much functionality in common with the ICTM coprocessor. As a result, by merging the ICTM coprocessor into the Patmos processor, a number of (expensive) registers and their required duplicates can be removed. The Delayed Triggering of Interrupts, described in [3, Section 2.7.5] is similar. As a result the overall Patmos processor + ICTM coprocessor area can be significantly reduced.

Interrupt Handling in Software

Saving and restoring of application timer and interrupt vectors as part of context switch, or provide a software-only application timer and interrupt vector save and restore to implement virtualised interrupts. This removes some of the ICTM coprocessor hardware for limited software complexity. Within the current interrupt IO module (and the associated API) of Patmos, interrupt vectors cannot be saved or restored. Instead, only individual interrupts can be acknowledged / re-raised, although masking interrupts is possible, as long as this is done before the interrupt is raised. Support for interrupts during the handling of other interrupts is already available, all register-based is pushed on/popped off the stack by prologue / epilogue software at the start and end of an interrupt handler.

Memory-Mapped I/O to Instructions

Instead of programming the ICTM coprocessor through the software API, which translates C function calls to memory-mapped I/O read and write instructions, dedicated Patmos instructions could be used, since the functionality is part of Patmos. Examples include functions listed in the Deliverable D2.4 *Design and Implementation of Interrupt Virtualisation Hardware and APIs* and Appendix A.

We expect that with these changes the ICTM coprocessor functionality is efficiently and effectively part of the Patmos processor. The T-CREST project has therefore already implemented some of these recommendations into account to reduce the dependence on the ICTM coprocessor, or to eventually even eliminate it. This may be seen from the addition to the April 2014 version of [3] of Section 2.7 Exceptions: Interrupts, Faults and Traps [3], in particular Section 2.7.5 Delayed Triggering of Interrupts [3] and Section 2.7.6 Sleep Mode [3] resulting in the May 2014 version of [3].

7 Requirements Evaluation from Deliverable D2.4

This section repeats the requirements and how they are satisfied from Deliverable D2.4 *Design and Implementation of Interrupt Virtualisation Hardware and APIs*.

We state nine general requirements on what the ICTM coprocessor achieves as the requirements on software and hardware architectures Deliverable D2.2 *Concepts for Interrupt Virtualisation*. The requirements are followed by a comment regarding the extent to which it is fulfilled by the ICTM coprocessor.

1. The RTOS must support multiple concurrent applications, each consisting of a set of communicating tasks.

   It can be supported by the inter- and intra-application schedulers in RTOS.

2. An application's tasks may be mapped on one or more processors.

   It is one of the assumptions of the T-CREST project and can be supported by the intra-application scheduler.

3. The RTOS must partition applications, in the sense that the worst-case and actual timing behaviours of any application are not affected by the absence or behaviour of any other application.

   It is supported by the time partitioning and interrupt virtualization mechanisms implemented in this document.

4. Each application must specify its own task scheduler, specific to its model of computation.

   It is supported by T-CREST platforms.

5. The following models of computation shall be supported: cyclo-static dataflow, Kahn process networks, time-triggered.

   Based on the interrupt virtualization techniques introduced in this document, cyclo-static dataflow and time-triggered models could be implemented and supported in OS schedulers on T-CREST platforms. Since Kahn process networks are not analyzable at design time, they are out the scope of the T-CREST research concerns.

6. A task scheduler may be non-preemptive (cooperative) or preemptive.

   Besides the support on cooperative scheduler, the developed device drivers and APIs provide time control on task level, which can be used by the task scheduler to implement preemptive scheduling.

7. A preemptive task scheduler shall be able to allocate a cycle budget to a task. The task will be preempted in the application when the cycle budget has been depleted.

   The intra-application scheduler can manage the cycle budget for tasks, based on the device drivers and APIs of the implemented ICTM coprocessor.

8. A preemptive task scheduler shall be able to allocate a deadline for a task. The task will be preempted at the latest time before the deadline when the application is active. The deadline shall be given as an absolute time.

   The intra-application scheduler can manage the deadline for tasks, based on the device drivers and APIs of the implemented ICTM coprocessor.

9. Each interrupt source, such as a hardware device, shall be assigned to and handled by one application. Interrupt arrival and handling shall not impede partitioning.

   The inter-application scheduler can exploit interrupt handling and virtualization mechanisms, based on the device drivers and APIs of the implemented ICTM coprocessor.

Here we list all requirements in aspect CORE and scope NEAR from Deliverable D 1.1 that are relevant for the processor work package. NON-CORE and FAR requirements are not listed here. The requirements are followed by a comment regarding the extent to which it is fulfilled by the research concept in this document.

P-0-505 Preemption:

The system shall provide means to implement preemption of running threads; these means shall enable an operating system or execution library to suspend a running thread immediately and make the CPU available to another thread.

The implemented interrupt structure has an interrupt management module to either forward external interrupts or generate timing dependent interrupts to preempt running threads.

P-0-506 Priority-preemptive scheduling:

The system shall provide means to implement CPU-local priority-preemptive scheduling without migration of threads between CPUs.

The implemented interrupt structure is independent of the scheduling algorithms adopted on processors. Therefore, priority-preemptive scheduling can be implemented.

P-0-508 Interrupts:

The CPU shall support interrupts.

Besides the original external interrupts and software interrupts, the developed device drivers and APIs can be used to keep track of the timing budget for applications and tasks, and trigger interrupts in scheduling.

P-4-014 Time control:

The processor shall support time control instructions. Such instructions are needed to implement interrupt virtualization.

The developed device drivers and APIs provide time control on both application and task levels, which can be used by the processor.

8 Conclusions

This document described the deliverable *D 2.7 Evaluation report on interrupt virtualization* of Work Package 2 of the T-CREST project, due 36 months after project start as stated in the Description of Work. The Interrupt Control and Timing Manager (ICTM) is the hardware accelerator of the Patmos processor [3] that implements required functionality in the T-CREST platform. This document describes how the ICTM coprocessor was integrated with the Patmos processor, from a hardware and software perspective. We evaluated the hardware architecture defined in the project. The main recommendations for the T-CREST project are to merge the ICTM coprocessor functionality into the Patmos processor, due to overlapping functionality and relatively high cost of the ICTM coprocessor. This functionality merging has to some extent already been effectuated in the project.

A ICTM coprocessor API

Updated ICTM coprocessor software API.

```
/*
 *
 * Copyright 2014 Eindhoven University of Technology
 */

//interrupt manager functions
/**
 * raise bits of the ICTM interrupt vector
 * ORs the given vector with the ICTM interrupt vector
 */
void hw_ictm_int_raise_vector(unsigned int vector);

/**
 * acknowledge bits of the ICTM interrupt vector
 * lowers the bits of the ICTM interrupt vector where the given vector has raised bits
 * generates an error if an interrupt is acknowledged that is not raised
 */
void hw_ictm_int_ack_vector(unsigned int vector);

/**
 * acknowledge bits of the ICTM interrupt vector
 * lowers the bits of the ICTM interrupt vector where the given vector has raised bits
 * does not generate an error: should only be used by microkernel
 */
void hw_ictm_int_lower_vector(unsigned int vector);

/**
 * get the interrupt vector of the ICTM, regardless of the mask
 */
int hw_ictm_int_get_vector();

/**
 * get the interrupt vector after ANDed with the mask
 */
int hw_ictm_int_get_masked_vector();

/**
 * sets the mask
 * interrupts are only propagated when the corresponding mask bit is raised
 */
void hw_ictm_int_set_mask(unsigned int mask);

/**
 * programs a system interrupt based on the system timer
 * the interrupt is generated at the given timevalue
 * if the value is in the past, it is generated immediately and generates a warning
 */
void hw_ictm_int_set_sysint_systimer_abs(TIME timevalue);

/**
 * programs a system interrupt based on the system timer
 * the interrupt is generated in the given timevalue, i.e. the interrupt is generated at the
     timevalue: (current system timer value + given timevalue)
 */
void hw_ictm_int_set_sysint_systimer_rel(TIME timevalue);

/**
 * programs a partition interrupt based on the partition timer
 * the interrupt is generated at the given timevalue
```

```
 * if the value is in the past, it is generated immediately and generates a warning
 */
void hw_ictm_int_set_parint_partimer_abs(TIME timevalue);

/**
 * programs a partition interrupt based on the partition timer
 * the interrupt is generated in the given timevalue, i.e. the interrupt is generated at the
     timevalue: (current partition timer value + given timevalue)
 */
void hw_ictm_int_set_parint_partimer_rel(TIME timevalue);

/**
 * programs a partition interrupt based on the system timer
 * the interrupt is generated at the given timevalue
 * if the value is in the past, it is generated immediately and generates a warning
 */
void hw_ictm_int_set_parint_systimer_abs(TIME timevalue);

/**
 * programs a partition interrupt based on the system timer
 * the interrupt is generated in the given timevalue, i.e. the interrupt is generated at the
     timevalue: (current system timer value + given timevalue)
 */
void hw_ictm_int_set_parint_systimer_rel(TIME timevalue);

/**
 * enables the partition interrupts based on the system timer
 */
void hw_ictm_int_enable_parint_systimer();

/**
 * disables the partition interrupts based on the system timer
 * when disabled, a partition interrupt based on the system timer is never generated until
     enabled
 */
void hw_ictm_int_disable_parint_systimer();

/**
 * get the time at which the next system interrupt based on the system timer is programmed
 */
TIME hw_ictm_int_get_next_sysint_raise_systimer();

/**
 * get the time at which the next partition interrupt based on the partition timer is programmed
 */
TIME hw_ictm_int_get_next_parint_raise_partimer();

/**
 * get the time at which the next partition interrupt based on the system timer is programmed
 */
TIME hw_ictm_int_get_next_parint_raise_systimer();

/** \} */


/**
 *\defgroup ICTM_systimer System Timer
 *
 * \brief system timer functions
 * \ingroup ICTM
 * \{
 */
/**
 * set the current value of the system timer
```

```
 */
void hw_ictm_systimer_set(TIME timevalue);

/**
 * get the current value of the system timer
 */
TIME hw_ictm_systimer_get();

/**
 * start the counting of the system timer
 * the system timer increases every system clock cycle
 */
void hw_ictm_systimer_start();

/**
 * stop the counting of the system timer
 * the system timer value remains the same
 */
void hw_ictm_systimer_stop();

/**
 * enable the system interrupts based on the system timer
 */
void hw_ictm_systimer_enable_int();

/**
 * disables the system interrupt based on the system timer
 * when disabled, a partition interrupt based on the system timer is never generated until
     enabled
 */
void hw_ictm_systimer_disable_int();

/**
 * set the system timer value and immediately start the timer
 * the system timer increases every system clock cycle
 */
void hw_ictm_systimer_set_start(TIME timevalue);

/**
 * set the period of the system interrupts
 * a new system interrupt is generated exactly the period length number of cycles after the
     previous interrupt
 * will only be used when interrupt period is enabled
 */
void hw_ictm_systimer_set_int_period(TIME timevalue);

/**
 * enable the period of the system interrupts
 * a new system interrupt is generated exactly the period length number of cycles after the
     previous interrupt
 */
void hw_ictm_systimer_enable_int_period();

/**
 * disable the period of the system interrupts
 * only one interrupt is generated when an interrupt is programmed
 */
void hw_ictm_systimer_disable_int_period();

/**
 * get the TIME value of the last generated system interrupt
 */
TIME hw_ictm_systimer_get_last_sysint();
```

```
/**
 * set the TIME value of the last generated system interrupt
 * only used for restoring context, as this value can only be read
 */
void hw_ictm_systimer_set_last_sysint(TIME timevalue);

/**
 * set the preload_par_context value
 * the ICTM is waked up this value of cycles before the next partition slot
 * in this time the context of the partition is loaded
 * when loading the context takes longer or shorter, this value needs to be corrected, otherwhise
     the partition executes less or more cycles respectively
 */
void hw_ictm_systimer_set_preload_par_context(int length);

/**
 * get the extra work of the system interrupt
 * the extra work is the number of partition cycles before the system interrupt handler is
    invoked
 */
int hw_ictm_systimer_get_sysint_extra_work();

/** \} */

/**
 *\defgroup ICTM_partimer Partition Timer
 *
 * \brief partition timer functions
 * \ingroup ICTM
 * \{
 */

/**
 * sets the current value of the partition timer
 */
void hw_ictm_partimer_set(TIME timevalue);

/**
 * get the current value of the partition timer
 */
TIME hw_ictm_partimer_get();

/**
 * start the counting of the partition timer
 * the partition timer increases every partition clock cycle
 */
void hw_ictm_partimer_start();

/**
 * stop the counting of the partition timer
 * the partition timer value remains the same
 */
void hw_ictm_partimer_stop();

/**
 * enable the partition interrupts based on the partition timer
 */
void hw_ictm_partimer_enable_int();

/**
 * disables the partition interrupt based on the partition timer
 * when disabled, a partition interrupt based on the partition timer is never generated until
    enabled
 */
```

```
void hw_ictm_partimer_disable_int();

/**
 * set the partition timer value and immediately start the timer
 * the partition timer increases every partition clock cycle
 */
void hw_ictm_partimer_set_start(TIME timevalue);

/**
 * set the period of the partition interrupts
 * a new partition interrupt is generated exactly the period length number of cycles after the
     previous interrupt
 * will only be used when interrupt period is enabled
 */
void hw_ictm_partimer_set_int_period(TIME timevalue);

/**
 * enable the period of the partition interrupts
 * a new partition interrupt is generated exactly the period length number of cycles after the
     previous interrupt
 */
void hw_ictm_partimer_enable_int_period();

/**
 * disable the period of the partition interrupts
 * only one interrupt is generated when an interrupt is programmed
 */
void hw_ictm_partimer_disable_int_period();

/**
 * set the sysint_stop_offset value
 * the sysint stop offset is the number of cycles between starting the interrupt handler and the
     hw_ictm_stop_partimer_at_start_mkslot (at the beginnning of the interrupt handler)
 * is used to revert the partition timer value to the time of the last executed instruction of
     the partition
 * should be updated when the start of the system interrupt handler is changed
 */
void hw_ictm_partimer_set_sysint_stop_offset(int offset);

/**
 * get the TIME value of the last generated partition interrupt
 */
TIME hw_ictm_partimer_get_last_parint();

/**
 * set the TIME value of the last generated partition interrupt
 * only used for restoring context, as this value can only be read
 */
void hw_ictm_partimer_set_last_parint(TIME timevalue);

/**
 * get the extra work of the partition interrupt
 * the extra work is the number of partition instructions executed before the partition interrupt
      handler is invoked
 */
int hw_ictm_partimer_get_parint_extra_work();

/**\}*/

/**
 *\defgroup ICTM_osfunc OS functions
 *
 * \brief OS functions
 * \ingroup ICTM
```

```
 * \{
 */


/**
 * get the error register, in which the last 8 bits represent (from left to right descending):
 * (7) a system interrupt occurs when the previous was not acknowledged
 * (6) acknowledging an unset interrupt
 * (5) not used
 * (4) not gated when preload should begin, indicating that the mk slot was not long enough
 * (3) denominator of 0
 * (2) system interrupt was raised before the end of the mk slot
 * (1) partition counter warning programmed in past
 * (0) system counter warning programmed in past
 */
int hw_ictm_get_error_reg();
```

B Platform configuration file

```
<patmos default="default.xml">
  <description>Configuration for the ML605 board with on-chip memory and ICTM</description>

  <frequency Hz="66000000" />

  <ExtMem size="1M" DevTypeRef="OCRam" />

  <IOs>
    <IO DevTypeRef="Uart" offset="8"/>
    <IO DevTypeRef="Leds" offset="9"/>
    <IO DevTypeRef="Ictm" offset="10" intrs="2"/>
  </IOs>

  <Devs>
    <Dev DevType="Uart" entity="Uart" iface="OcpCore">
      <params>
        <param name="baudRate" value="115200"/>
        <param name="fifoDepth" value="16"/>
      </params>
    </Dev>
    <Dev DevType="Leds" entity="Leds" iface="OcpCore">
      <params>
        <param name="ledCount" value="9"/>
      </params>
    </Dev>
    <Dev DevType="OCRam" entity="OCRamCtrl" iface="OcpBurst">
      <params>
        <param name="addrWidth" value="20" />
      </params>
    </Dev>
    <Dev DevType="Ictm" entity="Ictm" iface="OcpCore">
    </Dev>
  </Devs>
</patmos>
```

References

[1] Ashkan Beyranvand Nejad, Anca Molnos, and Kees Goossens. A software-based technique enabling composable hierarchical preemptive scheduling for time-triggered applications. In *Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2013.

[2] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, Andrew Nelsson, Jude Ambrose, and Kees Goossens. Design and Implementation of an Operating System for Composable Processor Sharing. *Microprocessors and Microsystems (MICPRO)*, 35(2):246–260, 2011.

[3] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.