T-CREST

TIME-PREDICTABLE MULTI-CORE ARCHITECTURE FOR EMBEDDED SYSTEMS

Project Number 288008

D 2.6 Integration and Report

Version 1.0
25 September 2014
Final

Public Distribution

Technical University of Denmark

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 José Neves Av. D. João II, Torre Fernão de Magalhães, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: jose.neves@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	1 September 2014
0.9	Version for partner review	24 September 2014
1.0	Final version	25 September 2014

Executive Summary

This document describes the deliverable *D 2.6 Integration and Report* 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 final version of Patmos including time-predictable cache designs and the integration of Patmos processors with the NoC and the memory tree.

1 Introduction and Background

Processors for future embedded systems need to be time-predictable *and* provide a reasonable worst-case performance. Therefore, we present the time-predictable processor Patmos as one approach to attack the complexity issue of WCET analysis. Patmos is a statically scheduled, dual-issue RISC processor that is optimized for real-time systems. Instruction delays are well defined and visible through the instruction set architecture (ISA). This design simplifies the WCET analysis and helps to reduce the overestimation caused by imprecise information.

The dual-issue pipeline with specially designed caches provides good single thread performance. We provide a chip-multicore version of Patmos as a time-predictable execution platform for multi-threaded applications.

A major challenge for the WCET analysis is the memory hierarchy with multiple levels of caches. We tackle this issue through caches that are especially designed for WCET analysis. For instructions we adopt the method cache [24], which operates on whole functions/methods and thus simplifies the modeling for WCET analysis. Furthermore, we propose a split-cache architecture for data [25, 29], offering dedicated caches for the stack [1] and for other data, as well as a scratchpad memory.

Aside from the hardware implementation of Patmos, we also present the integration with the compiler for the development of future real-time applications. Patmos is designed to facilitate WCET analysis; its internal operation is thus well defined in terms of timing behavior and explicitly made visible on the ISA level. Features that are hard to predict are avoided and replaced by more predictable alternatives, some of which rely on the (low-level) programmer or compiler to achieve optimal results, i.e., low actual WCET and good WCET bounds. We provide a *WCET-aware* software development environment tightly integrating WCET tools and compilers [23, 5].

The processor and its software environment is intended as a platform to explore various time-predictable design trade-offs and their interaction with WCET analysis techniques as well as WCET-aware compilation. We propose the co-design of time-predictable processor features with the WCET analysis tool, similar to the work by Huber et al. [14] on caching of heap allocated objects in a Java processor. Only features where we can provide a static program analysis shall be added to the processor.

Figure 1 shows the integration of Patmos into the T-CREST the multicore platform. Several processor cores, the Patmos processors [30], are connected to two NoCs: (1) a core NoC for message passing between processor-local scratchpad memories [17, 31, 27], and (2) a memory NoC [9, 28] that connects all processor cores to the shared, external memory via the memory controller.

Patmos, the NoC, and its tool chain are open source; the hardware implementation is available under the BSD license.[1] Deliverable D 8.5 (Open Source Reference Implementation) gives details where to find the T-CREST components. Detailed descriptions of the instruction set and the build process are available in the Patmos handbook [26].

[1] see: `https://github.com/t-crest/patmos`

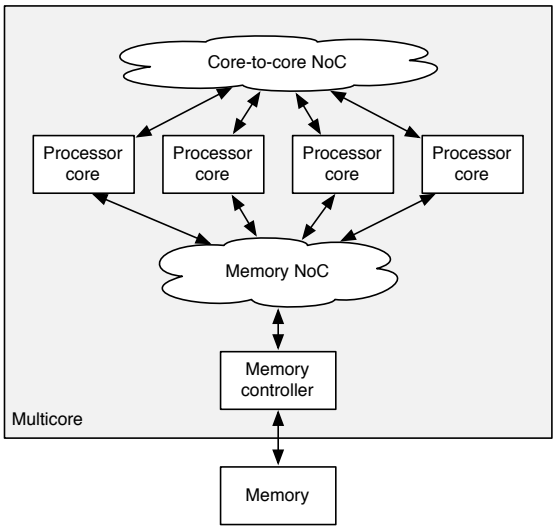


Figure 1: T-CREST multicore architecture with several processor cores connected to two NoCs: one for core-to-core message passing and one for access to the shared, external memory.

2 The Architecture of Patmos

Patmos is a 32-bit, RISC-style microprocessor optimized for time-predictable execution of real-time applications. In order to provide high performance for single-threaded code, a two-way parallel architecture was chosen. Patmos is configurable as two-way pipeline or as single-way pipeline to save resources. For multi-threaded code we provide a chip-multiprocessor system with statically scheduled access to shared main memory.

Patmos is a statically scheduled, dual-issue RISC microprocessor. All instruction delays are thus explicitly visible at the ISA-level, and the exposed delays from the pipeline need to be respected in order to guarantee correct code. However, knowing all pipeline delays and the conditions under which they occur simplifies the processor model required for WCET analysis and helps to improve accuracy.

The modeling of memory hierarchies with multiple levels of caches is critical for practical WCET analysis. Patmos simplifies this task by offering caches that are especially designed for WCET analysis. Accesses to different data areas are quite different with respect to WCET analysis. Static data, constants, and stack allocated data can easily be tracked by static program analysis. Heap allocated data on the other hand demands for different caching techniques to be analyzable [14]. Therefore, Patmos contains two data caches, one for stack cache and one for other data.

2.1 Fully Predicated Instruction Set

It instruction set is based on a RISC style load/store instruction set and take at most three register operands. However, in contrast to common RISC architectures, all instructions are fully predicated. While control-flow instructions and instructions that access memory can be executed only in the first pipeline, arithmetic and logic instructions can be executed in both pipelines.

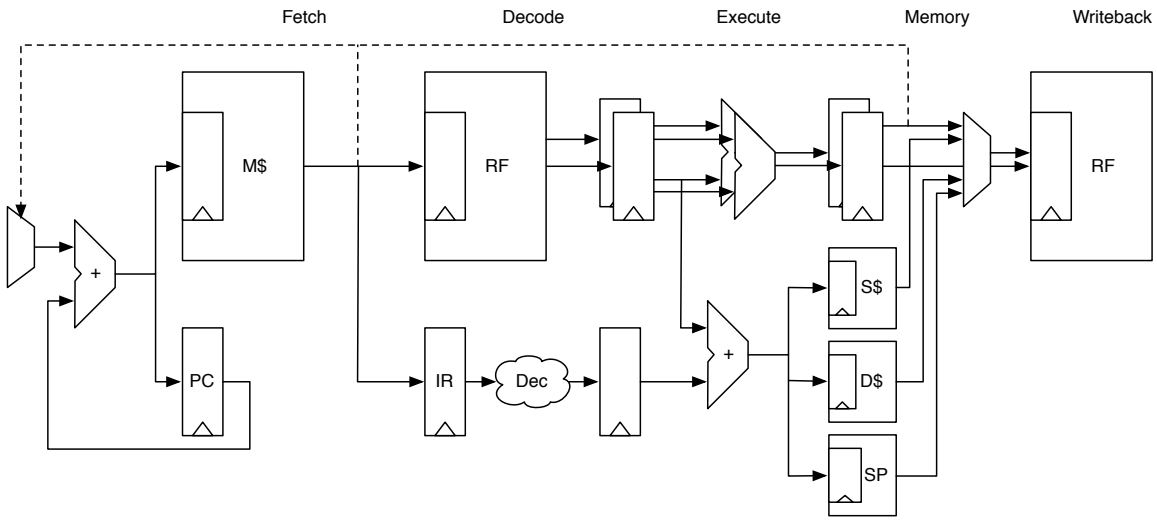


Figure 2: Dual issue pipeline of Patmos with fetch, decode, execute, memory, and write back stages.

The first instruction of an instruction bundle contains a bit to encode the length of the bundle (32 or 64 bits). Register addresses are at fixed positions to allow reading the register file parallel to instruction decoding. The main pressure on the instruction coding comes from size of constant fields and branch offsets. However, as we support fetching of up to two 32-bits words for the dual-issue pipeline, we use this feature to support ALU operations with 32-bit constants. The constant is encoded in the second instruction slot. This enables loading 32-bit constants in a single cycle. Furthermore, most ALU instructions can be performed with a 12-bit constant operand, which saves space and leaves the second instruction slot free for other instructions. Branches (conditional and unconditional) are relative with a 22-bit offset. Function calls to a 32-bit address are supported by a register-indirect call instruction.

To reduce the number of conditional branches and to support the single-path programming paradigm [21, 22], Patmos supports fully predicated instructions. Predicates are set with compare instructions, which can be predicated themselves. Patmos has compare instructions between registers and against constant 0. Patmos has 8 predicate registers.

Access to the different types of data areas are explicitly encoded with the load and store instructions. This feature helps the WCET analysis to distinguish between the different data caches. Furthermore, it can be detected earlier in the pipeline which cache will be accessed.

2.2 Dual-Issue Pipeline

Patmos contains 5 pipeline stages: (1) instruction fetch (`FE`), (2) decode and register read (`DEC`), (3) execute (`EX`), (4) memory access (`MEM`), and (5) register write back (`WB`). Figure 2 shows an overview of Patmos' pipeline.

The register file with 32 registers is shared between the two pipelines. Full forwarding between the two pipelines is supported. The basic features are similar to a standard RISC pipeline. The data

cache is split into different cache areas. The distinction between the different caches is performed with typed load and store instructions.

Figure 2 shows an overview of Patmos' pipeline. To simplify the diagram, forwarding and external memory access data paths are omitted. The method cache (M$), the register file (RF), the stack cache (S$), the data cache (D$), and the scratchpad memory (SP) are implemented in on-chip memories of an FPGA. All on-chip memories of Patmos use registered input ports. As the memory-internal input registers cannot be accessed, the program counter (PC) is duplicated with an explicit register. The instruction fetched from the method cache is stored in the instruction register (IR) and also used in the register file to fetch the register values during the decode stage.

Due to the dual issue pipeline the register file needs four read ports and two write ports. Such a memory is not available in an FPGA. One possibility is to double-clock an on-chip memory for two write ports and duplicate to provide 4 read ports [30]. In the current design we implement the register file with FPGA registers and use multiplexers for the read ports.

As Patmos provides full forwarding from both pipelines, this forwarding network consumes a lot of resources. If the full power of dual issue is not needed, Patmos can be configured as single-issue pipeline.

2.3 Local Memories and Caches

Patmos contains several caches (method, data, and stack cache) and scratchpad memories (SPM) for data and instructions. All caches are configurable in the size and all of them are optional and can be disabled. To distinguish between the different caches, Patmos implements typed load and store instructions. The type information is assigned by the compiler (for the stack cache) or by the programmer (for a data SPM).

2.3.1 Boot ROM and Scratchpad Memories

To bootstrap the processor it contains on-chip ROMs for instructions and data. Furthermore, it contains a small SPM dedicated to bootstrapping, such that small applications and test cases do not need to access external memory. For larger application, the boot ROM contains a boot loader that loads the application from a non-volatile memory or during development from a serial line into the main memory.

Patmos also contains (optional) SPMs for instructions and data. These SPMs can be used in addition to caches or instead of caches, when code and or data caching shall be under program control.

2.3.2 Caches

Patmos contains three caches: a data cache, a method cache [6], and a stack cache [1].

The current implementation of the data cache is a direct mapped cache with write-through and no allocation on a write. Write-through was chosen as state-of-the-art WCET analysis tools do not track the state of a dirty bit in a write-back cache and therefore assume the worst case that a cache miss also

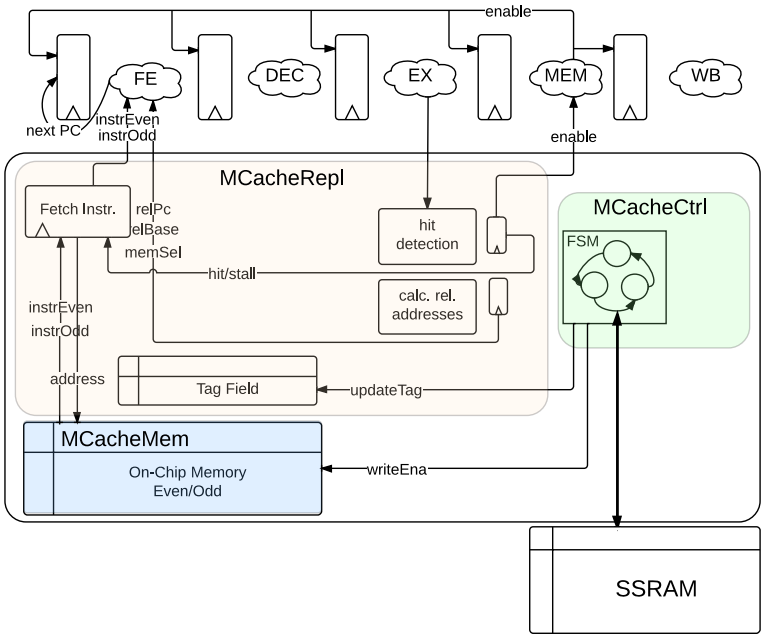


Figure 3: The method cache and the relation to the processor pipeline and the external memory

needs a write back of this cache line. The design decision to use a write-through policy is another example how WCET analyzability influences the hardware design for a time-predictable processor. For statically unknown load and store addresses Patmos has load and store instructions that bypass all caches.

The stack cache and the method cache are described in more detail in the following sections.

2.3.3 Miss Detection and Pipeline Stalling

The cache configuration of Patmos is special with respect to miss detection: for all three caches, misses are detected (and the pipeline stalled) in the memory stage. This is normal for a data cache, but a standard instruction cache misses in the fetch stage. However, the method cache performs miss detection just on call and return. Therefore, these instructions can stall as well in the memory stage.

The consequence of a single stalling pipeline stage is twofold: (1) the hardware implementation of stalling is simplified and (2) cache analysis becomes simpler. No two instructions can trigger a cache miss in the same clock cycle for two caches. We consider this feature to contribute to a timing-composable architecture. Different caches can be analyzed independently.

2.4 The Method Cache

Figure 3 gives an overview of the method cache structure. The figure shows the interaction between the pipeline stages of the processor and the sub-components of the method cache. Furthermore, the external memory (in our implementation an SSRAM) is shown as main memory, the source for function loads. Patmos fetches instructions in the fetch stage, while it detects misses and stalls the

pipeline in the memory stage. Detecting all possible misses in the same pipeline stage has the benefit that only one miss is outstanding. If two instructions can miss in different stages, the WCET analysis might sometimes need to be conservative and assume a longer latency for an instruction cache miss due to a concurrent data cache miss.

The method cache acts in parallel to the processor pipeline. The on-chip memory that contains instructions is conceptually located in parallel to the fetch stage, while the hit detection is in parallel to the execute stage. The pipeline is stalled from an instruction in the memory stage. If a function has to be loaded from external memory into the cache, the processor pipeline is stalled by the memory transfer unit, which is shown as a finite state machine (FSM) in the figure.

2.4.1 Relative Addressing

Loading full functions on a cache miss enables relative addressing within the cache. The relative address is calculated after each call or return. If a new function is called, the program counter is set to the position of the function in the on-chip memory, relative to the on-chip memory's base address. This relative addressing simplifies the hardware implementation, because there is no need to translate between the absolute address of an instruction and its location in the on-chip memory.

When returning to a function, the method cache needs the function's base address to perform hit detection and load the function into the on-chip memory on a miss. Furthermore, the instruction to return to must be identified. For this purpose we use the offset of the return location relative to the function's base address. The value of the program counter after a return is then the function's position in the on-chip memory plus this offset.

2.4.2 Hardware Implementation

We have implemented the method cache in Chisel [3], an object-oriented hardware-construction language. Therefore, hardware components are class instances.

The main class `MCache` serves as a top-level component and instantiates and interconnects all subcomponents of the method cache. Furthermore, `MCache` contains the interface to the Patmos processor pipeline. The following sub-components implement the method cache:

- `MCacheMem` implements the on-chip memory,
- `MCacheRepl` implements the underlying replacement strategy, and
- `MCacheCtrl` implements the state machine, which controls the transfer from external memory.

The `MCacheMem` component implements on-chip memory, which stores the instructions of the cached functions. The implementation considers the underlying dual-issue pipeline, which fetches one or two instructions in every clock cycle. Therefore, the on-chip memory is split into two memories: the "even" or the "odd" memory for even and odd addresses. This memory structure allows to fetch two instruction words concurrently, even if they are not aligned to a double-word boundary.

The `MCacheRepl` component implements the replacement strategy for the method cache. Since different strategies can be used for function replacement, we propose different classes for the individual

replacement policies. Each loaded function has an entry in the tag memory. A function is identified inside the tag memory by the base address (i.e., the start address of the function), which is provided on calls and returns.

Furthermore, the `MCacheRepl` component performs the hit/miss detection when a call or return instruction is executed. The entries of the tag field memory have to be searched for a valid address. If a hit is detected, the position of the function in the cache is used to compute the new program counter. A miss leads to a stall of the pipeline and the component waits until the function is loaded into the cache. When a function is loaded, the update of the tag field is signalized by the `update_tag` signal from the control unit. The base address is then written to the tag field at the replaced index. An additional state machine handles the invalidation of tag fields for functions that are overwritten by the new function.

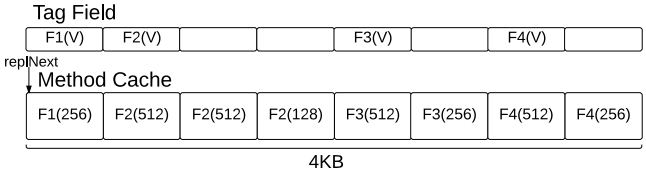
The method cache is also responsible for calculating a new relative address on every call or return. This is done by the `MCacheRepl` component, which calculates the new program counter using the base address from the execute stage and the position of the function in the cache. After the call or return is executed, the value is provided to the fetch stage and the execution continues with the updated program counter.

The `MCacheCtrl` component implements the transfer from the external memory to the on-chip cache memory. A finite state machine (FSM) handles the correct sequence of communication between the external memory controller and the method cache. As long as the replacement unit reports a hit, the controller stays idle. A miss causes the unit to load the requested function from the connected external memory. A subset of the OCP protocol [**?**] is used as an interface between the method cache and the memory controller. The first state of the FSM requests the size of the function, which is located immediately before a function's instructions. The OCP protocol allows burst reads from addresses that are aligned to the burst size. Therefore, the function size is not always the first word in the received burst. As soon as the size is loaded, subsequent words are written to the on-chip memory through the `MCacheRepl` component. The transfer state requests new burst reads until the function is fully loaded into the cache. The processor pipeline is stalled during the whole transaction and resumes execution when the function is loaded.

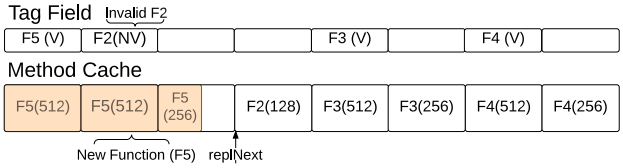
2.4.3 Method Cache Variations

We have implemented two variants of the method cache: (1) the version with fixed block sizes, similar to the original JOP method cache, and (2) a version with variable-size method allocation. Both variations use a FIFO replacement strategy.

Fixed-Block Method Cache A simple implementation of a replacement strategy for the method cache is to use FIFO replacement when a new function has to be moved to the cache. According to the FIFO policy, the new function replaces the oldest entry in the cache. In practice, the implementation requires only a pointer to the next block to be replaced. The cache operates like a ring buffer and



(a) Method cache filled with functions F1-F4



(b) F5 replaces F1 and F2

Figure 4: Fixed-block method cache with FIFO replacement

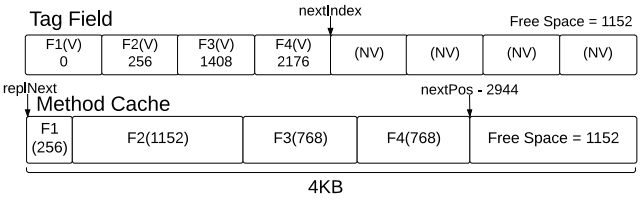
starts again at the head when the pointer overruns the cache size. The update of the replace position can be described by the following formula:

$$NewPos = (OldPos + \#UsedBlocks \times BlockSize) \mod M\$Size$$

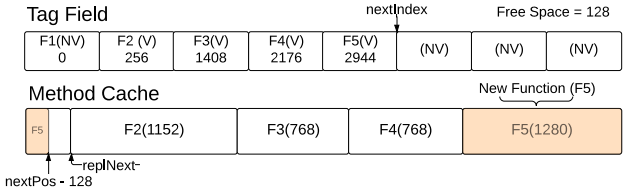
The on-chip memory used for the cache is divided into blocks with a fixed size. The tag memory has one entry for each block. Therefore, the maximum number of cached functions is limited by the number of blocks. In our implementation we allow a function to span several blocks. In case a function uses more than one block, the associated function tag is stored in the first block. All tag fields of the additional blocks are invalidated, since the old functions are overwritten by the new content.

Figure 4 shows an example of a fixed-block method cache and the associated tag field where FIFO replacement is applied to fixed blocks. The cache in this example is 4 KB large and consists of eight blocks, each being 512 bytes large. The method cache is already filled with four functions (a) where the values in parentheses state the occupancy of each block. Consider a call that calls function F5, which requires three consecutive blocks. The code for F5 replaces all of F1 and part of the code of F2 (b); the tag for F1 is overwritten with the tag for F5. The tag field for F2 must be invalidated, since only a fragment of the function remains in the cache. This depicts a drawback of a block arrangement. The cache is never fully utilized and the function allocation is limited by a fixed block structure. Therefore, a good tradeoff between cache size, block size and maximal function size has to be chosen. We explore different cache configurations in the evaluation section.

Variable-Size Method Cache A basic problem of the replacement with fixed blocks is the waste of memory in blocks that are not fully filled. This problem arises especially when the function sizes vary greatly and the cache could hold more function tags but is limited by functions wasting unused space in the memory. A solution is to build a more flexible cache structure in which functions are not stored into blocks of a predefined size, but are allocated variably in the cache memory. We



(a) Method cache filled with functions F1-F4



(b) F5 fills free space and replaces F1

Figure 5: Variable-size method cache with FIFO replacement

still use a FIFO replacement strategy, but in order to provide a tight arrangement of the functions in the cache we place them back-to-back. A function can be located at any address in the cache with the restriction of a double word address alignment. In a similar way to a fixed-block replacement, the cache operates like a ring buffer and an overflowing function rolls over the ending and starting address of the memory. The new replace position is updated as follows:

$$NewPos = (OldPos + FunctionSize) \bmod M\$Size$$

In a fixed-block method cache, positions in on-chip memory are implicitly tied to particular entries in the tag memory. In contrast, a variable-size method cache has to explicitly store a function's position in the on-chip memory. Furthermore, we have to keep track of the currently available space in the cache and the function sizes. If a function is replaced in the cache, the size is freed for the new function. A pointer is used to track the tag field that is going to be replaced next. If a function requires more space than actually allocated by the replaced function, further functions have to be overwritten and their tag fields have to be invalidated. This is done by a second state machine that sequentially invalidates all further address tags until enough space is available. As the functions have to be loaded into the cache as a whole, a gap of free space between the end of the loaded function and the function to be replaced next will probably arise. This free space is tracked and integrated into the calculation of the available space for the next replacement.

Figure 5 shows an example of a variable-size method cache. The cache is 4 KB large and its tag memory can hold up to eight references to functions. As in the example in Figure 4, four functions, F1 to F4, are already loaded into the cache (a). In contrast to the example in Figure 4, the cache still has a free space of 1152 bytes without replacing currently loaded functions. An invocation of function F5 fills another tag field and recalculates the free space (b). Since the function size overflows the current free space, the next function in the FIFO buffer (signalized by the *replNext* pointer) is invalidated and replaced by the new instruction code. This example points out the advantage of a variable-sized block compared to the scenario in Figure 4, where two functions in the cache have to be replaced.

Table 1: Hardware costs of a 4 KB method cache in logic cells

# functions	LC (V)	LC (F)
4	917	671
8	1199	806
16	1764	1035
32	2846	1566

2.4.4 Hardware Cost

Table 1 shows the hardware cost for the variable-size method cache (V) and the fixed-block method cache (F) for a 4 KB method cache for different number of functions/blocks. The table shows the number of logic cells (LC), the basic building blocks in an FPGA, for the method cache. Additionally 4 KB of on-chip memory is needed. We see that the resource consumption increases considerably with the number of functions handled. This is expected as the hardware implements a fully associative lookup structure for the tag memory.

The variable-size method cache needs additional tag memory to store the current position of a function in the cache and the size of the function. This explains the higher hardware costs compared to a fixed-block method cache.

To set these numbers in context, a dual issue Patmos, including the method cache and a memory controller, consumes 12426 LCs and 20 KB of on-chip memory. The dual-issue pipeline of Patmos alone consumes 9155 LCs and a single-issue pipeline of Patmos 3317 LCs. The dual-issue pipeline is larger because it uses dedicated registers for the register file and supports full forwarding between the two pipelines.

2.5 The Stack Cache

The stack cache is a processor-local, on-chip memory. The stack cache operates similar to a ring buffer. It can be seen as a stack-cache-sized window into the main memory address range. To manage the stack cache, we use three additional instructions: `reserve`, `ensure`, and `free`. Two hardware registers define which part of the stack area is currently in the stack cache.

2.5.1 Stack Cache Manipulation

We present the mechanics of the stack cache in C code for easier readability. However, the hardware implementation is a synchronous design and the algorithm is implemented by a state machine that handles the memory spill and fill operations. In the C code following data structures are used:

`mem` is an array representing the main memory,

`sc` is an array representing the stack cache,

`m_top` is the register pointing to the top of the saved stack content in the main memory, and

```
void reserve(int n) {

    int nspill, i;

    sc_top -= n;
    nspill = m_top - sc_top - SC_SIZE;
    for (i=0; i<nspill; ++i) {
        mem[m_top] = sc[m_top & SC_MASK];
        --m_top;
    }
}
```

Figure 6: The `reserve` instruction provides `n` free words in the stack cache. It may spill data into main memory.

`sc_top` points to the top element in the stack cache.

The two pointers are full-length address registers. However, when addressing the stack cache, only the lower n bits are used for a stack cache of a size of 2^n words. The constant `SC_SIZE` represents the stack cache size and `SC_MASK` is the bit mask for the stack cache addressing. The stack cache is managed in 32-bit words. Therefore, the pointers count in 32-bit words.

At program start the stack cache is empty and both pointers, `m_top` and `sc_top`, point to the same address, the address that is used for the stack area. `m_top` points to the next not-yet-used word in main memory. Similar, `sc_top` points to the next unused word of the stack cache. Therefore, the number of currently valid elements in the stack cache is `m_top` - `sc_top`.

The compiler generates code to grow the stack downward, as it is common for many architectures. Growing the stack downwards has historical reasons. However, for multi-threaded systems each thread needs a reserved, fixed memory area for the stack and there is no benefit from growing the stack downwards.

Reserve The `reserve` instruction, as shown in Figure 6, reserves space in the stack cache. Typed load and store instructions use this reserved space. The reserve instruction may spill data to the main memory. This spilling happens when there are not enough free words in the stack cache to reserve the requested space.

The processor reads the number of words to be reserved (the immediate operand of the instruction) in the decode stage. The processor adjusts the `sc_top` register in the execution stage and also computes how many words need to be spilled in the execution stage. The processor spills to the main memory in the memory stage, as shown by the for loop in Figure 6.

Free The `free` instruction frees the reserved space on the stack. It does not fill previously spilled data back into the stack cache. It just changes the top of the stack pointer and may change the top of the memory pointer, as shown in Figure 7.

```
void free(int n) {

    sc_top += n;
    if (sc_top > m_top) {
        m_top = sc_top;
    }
}
```

Figure 7: The `free` instruction drops `n` elements from the stack cache. It may change the top memory pointer `m_top`.

```
void ensure(int n) {

    int nfill, i;

    nfill = n - (m_top - sc_top);
    for (i=0; i<nfill; ++i) {
        ++m_top;
        sc[m_top & SC_MASK] = mem[m_top];
    }
}
```

Figure 8: The `ensure` instruction ensures that at least `n` elements are valid in the stack cache. It may need to fill data from main memory.

Ensure Returning into a function needs to ensure that the stack frame of this function is available in the stack cache. The `ensure` instruction, as shown in Figure 8, guarantees this condition. This instruction may need to fill back the stack cache with previously spilled data. This happens when the number of valid words in the stack cache is less than the number of words that need to be in the stack cache. Filling the stack cache is shown in the loop in Figure 8.

One processor register serves as stack pointer and points to the end of the stack frame. Load and store instructions use displacement addressing relative to this stack pointer to access the stack cache.

2.5.2 Function Call Example

We illustrate the stack cache with an example of three functions: A calls B, which in turn calls C. Figure 9 shows the mapping of the stack cache to the main memory and the spilled content of the stack area in the main memory when the program is in function C. The blocks A, B, and C in the figure are the stack frames for functions A, B, and C. The stack mapping shows the stack usage as it would be without a stack cache. The main memory figure shows which parts of the stack frames have been spilled into the main memory. And the middle box (S$) shows the content of the stack cache. We can see that stack frame A is fully spilled to main memory. Frame B is partially spilled and it also wraps around in the stack cache. Stack frame C is fully in the stack cache.

The program in this example starts with an empty stack cache. After entering function A, its stack frame is allocated using a reserve instruction. This stack frame is represented as block A in Figure 9.

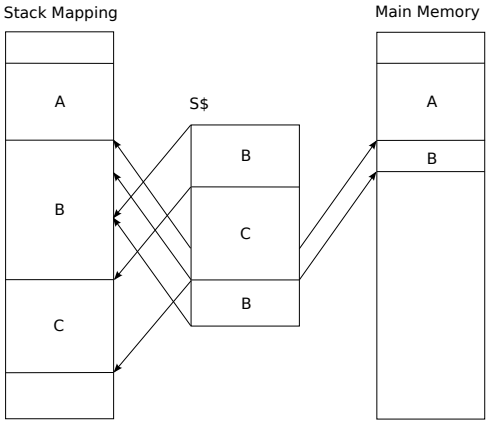


Figure 9: Stack mapping, the stack cache content, and the spilled stack area.

During execution of function A, function B is called. As before, B's stack frame is allocated by executing a reserve instruction, which checks whether the required space is available. Assuming that there is not enough free space in the stack cache, the stack frame of A is partially spilled to main memory. Then, another function call to C (Figure 9) causes the rest of the A's stack frame and a part of B's stack frame to be spilled to main memory. Before returning from C, its stack frame is freed. Since B's stack frame was partially spilled, an ensure instruction reloads the spilled parts of B's stack frame back into the stack cache. Similarly, the whole stack frame of A is reloaded after returning from B. Before returning from A, all the allocated space on the stack cache is freed and the stack cache becomes empty again.

2.5.3 Hardware Implementation

We implemented the stack cache in hardware for the processor Patmos [30], extended the cycle-accurate software simulator of Patmos to support the stack cache, and adopted the LLVM-based compiler for Patmos to make use of the stack cache.

To support the stack cache in Patmos, we added the required functionality to the execution and the memory stages. The execution stage computes new values for the stack cache registers and the condition if a spill or fill is needed. The memory stage contains the stack cache and performs the normal load and store operations. Furthermore, the memory stage contains the state machine for the spill and fill. On a fill or spill, the rest of the memory stage stalls the rest of the pipeline. In the following we describe the changes in the pipeline in more detail.

According to the algorithms introduced in Section 2.5, three instructions manage the stack cache. We extended the decode stage to support these instructions.

The execution stage performs the necessary computations to update the two stack cache registers. If necessary, the execution stage determines also the number of words that the memory stage will spill of fill. The execution stage sets the spill/fill signals for the memory stage.

In case of spill/fill operations (triggered by spill/fill signals from the execution stage), the memory stage stalls the other pipeline stages until the data transfer to/from main memory is completed. A state machine manages the spill/fill operations in the memory stage. This state machine keeps track

of the number of spilled/filled bytes and deactivates the stall signal after all the bytes are spilled/filled. Furthermore, based on the current state of the state machine, the state machine adjusts the pointer to the top of the memory to access different words of the main memory and the stack cache on each spill/fill operation. There are two different types of transfer to/from the stack cache: (1) one directly through the load and store instructions, and (2) one caused by fill/spill state machine. Thus a multiplexer in the memory stage determines if the input data for the stack cache is coming from the main memory or from a normal store instruction. Similar, the write path to the main memory is multiplexed between normal store instructions and the stack cache spill operation.

2.6 Implementation

A software simulator of the architecture is the first, important step to explore ideas and to serve as a reference design for the compiler and the hardware design. Therefore, we provide a software simulator and a hardware implementation of Patmos.

2.6.1 Simulator

At the start of the development of Patmos, a cycle-accurate software simulator was developed (as described in deliverable D 2.1, Software Simulator of Patmos). This simulator serves as the reference for the hardware implementation of Patmos, for the development of the compiler, and the porting of real-time operating systems. Furthermore, the simulator provides variants of caches and memory controller models and can thus be used for design-space exploration of caches.

Due to its focus on high-level simulation without modeling the underlying hardware in detail, architecture variations such as different method cache designs and spilling strategies can be quickly implemented. The simulator of Patmos thus serves well for quick evaluation of different design options.

2.6.2 Hardware Implementation

We use Chisel [3] for the implementation and simulation of the core design. Chisel was developed at the UC Berkeley and is a hardware-construction language, embedded in the programming language Scala. Consequently, Chisel allows the programmer to design efficient hardware components in a high level language. Scala, and therefore Chisel, are object-oriented and functional languages, enabling hardware design in an object-oriented way.

The Chisel back-end can generate both Verilog and C++ code. While Verilog is used to implement a design on an ASIC or FPGA, the C++ code implements a fast high-level simulation of the hardware and provides a test environment. We call the Chisel-generated C++ simulation the *emulator*, to distinguish it from the software-based simulator.

We adapted the top-level class of the emulator and added a model of an external SRAM memory. Furthermore, an executable file can be loaded into the memory (or optionally into the scratchpad memory of the model) to start the execution. The emulator produces a precise model of the system behavior and furthermore adds the possibility to easily insert debug information for explicit testing.

Since the Patmos emulator is auto-generated from the hardware description, it lends itself to high-level debugging and testing of the processor implementation. In contrast to the simulator, the individual registers and signals of the hardware design are emulated. We use the emulator to verify the cycle-accurate behavior of the Patmos simulator.

The use of Chisel facilitates the configuration of the hardware implementation. Cache sizes, the number of pipelines (dual- or single-issue), and other features can be controlled through a single XML configuration file. Also, I/O devices can be added to the processor through this configuration file. Parsing the configuration is done in Chisel, such that no code generation steps or manual editing of the code are necessary.

2.6.3 Co-Simulation

When building a complex hardware, such as the Patmos processor, testing, having good test coverage, and actually checking the outcome of the tests is very important. The software simulator can serve as gold reference for the hardware implementation of Patmos.

We compare on a cycle-by-cycle base the execution of the simulator and the execution of the emulator. To compare the two simulations we consider the most important state of a processor: the register file. A difference in other program visible state (program counter, predicate registers) might also be interesting, but a difference there will at some point (some cycles later) show up in the register file - if not, the failure would not be visible during a normal program execution.

A collection of assembler programs is co-simulated automatically every night.

2.6.4 Testing and Validation

Apart from a small set of test cases written in assembly, we use an extensive test suite with test cases written in C for testing and validation. This ensures that the compiler, the simulator, and the hardware implement the ISA consistently. The test suite includes the MiBench[2] and the Mälardalen[20] benchmarks. Furthermore, the test suite includes the `gcc.c-torture/execute` test cases from GCC's test suite, which cover a wide range of corner cases for compilation and execution. In total, the test suite contains more than 1000 test programs, yielding to more than 2000 individual test cases. The test suite is executed every night, such that regressions that might occur when evolving Patmos are detected quickly.

2.7 Memory Mapping and I/O Devices

2.7.1 Local and Global Address Space

The typed loads of Patmos imply two address spaces: a local address space that is accessed through local loads and stores, and a global address space that is accessed when using other access types. All caches use memory that is mapped to the global address space as backing memory. For example,

[2] `http://www.eecs.umich.edu/mibench/`

Address	Memory area
0x00000000–0x0000ffff	Data Scratchpad Memory
0x00010000–0x0001ffff	Instruction Scratchpad Memory (write only)
0xe0000000–0xe7ffffff	NoC interface configuration registers
0xe8000000–0xefffffff	NoC communication memory
0xf0000000–0xffffffff	I/O devices

Table 2: Address mapping for local address space

Address	Memory area
0x00000000–0x0000ffff	Boot Instruction ROM (only for code)
0x00010000–0x0001ffff	Instruction Scratchpad Memory (only for code)
0x00000000–0x7fffffff	External SRAM
0x80000000–0x8000ffff	Boot Data ROM (only for data)
0x80010000–0x8001ffff	Boot Data Scratchpad Memory (only for data)

Table 3: Address mapping for global address space

the data cache fetches data from global memory on a cache miss, and the stack cache uses global memory for spilling and filling. Consequently, there are two memory maps, one for the local address space and one for the global address space. Tables 2 and 3 show the respective address mappings. To simplify address decoding, the top four bits (A31–A28) are generally used to distinguish between different memory and I/O areas. The address range for I/O devices is divided further to distinguish the different devices, as discussed in Section 2.7.3.

As `call`, `ret`, and `brcf` do not include memory type information, the distinction between memory areas for these instructions is done solely through the address mapping. The boot instruction ROM and the instruction scratchpad memory are mapped to the lowest 128K of the global address space. Note that this applies only to these instructions; i.e., a `call` to address `0x00010100` executes code that is located in the instruction scratchpad, while non-local loads or stores to the same address access the external SRAM. Therefore, a binary that is loaded to external memory can use the lowest 128K of memory for data segments, but not for code segments.

The global address space also includes a ROM and a scratchpad for booting. The boot data ROM enables boot programs to use initialized data segments. By copying (parts of) the boot data ROM to the boot scratchpad, these programs can also use initialized data segments that require write access. Note that these memory areas can be accessed by loads and stores only; it is not possible to execute code located in them.

2.7.2 Boot Memories

By convention, the first four words of the boot data ROM contain information about data to be copied to the boot data scratchpad before starting actual execution. Table 4 shows the respective data fields.

Address	Name	Description
0x80000000	src_start	Start address of data to be copied
0x80000004	src_size	Size of data to be copied
0x80000008	dst_start	Destination for copying
0x8000000c	dst_size	Size of initialized data

Table 4: Boot data initialization information

Address	I/O Device	read	write
0xf0000000	cpuinfo	processor ID	–
0xf0000004	cpuinfo	clock frequency (Hz)	–
0xf0000100	excunit		
...	excunit	Exception unit and cache control	
0xf00001ff	excunit		
0xf0000200	timer	clock cycles (high word)	cycle interrupt time (high word)
0xf0000204	timer	clock cycles (low word)	cycle interrupt time (low word)
0xf0000208	timer	time in μs (high word)	μs interrupt time (high word)
0xf000020c	timer	time in μs (low word)	μs interrupt time (low word)
0xf0000800	UART	status	control
0xf0000804	UART	receive buffer	transmit buffer
0xf0000900	LED	–	output register
0xf0000a00	Keys	input register	–

Table 5: I/O devices and registers

Upon start, the program should copy `src_size` bytes of data from `src_start` to `dst_start`. If `dst_size` is greater than `src_size`, the remaining bytes are filled with zeroes.

2.7.3 I/O Devices

Each processor contains a minimum set of standard I/O devices, such as: processor ID, cycle counter, timer, and interrupt controller. For a minimum communication with the outside world a processor shall be attached to a serial port (UART). The UART represents `stdout`.

Within the I/O device memory area bits 11–8 are used to distinguish between different devices. I/O device registers are mapped and aligned to 32-bit words. If a register is shorter than a word, the upper bits shall be filled with 0 on a read. With this mapping each I/O device can have up to 64 32-bit registers.

In the initial prototype of Patmos we have 3 I/O devices: a system device that contains cycle and microsecond counters, a UART for basic communication, and LEDs on the FPGA board. One counter ticks with the clock frequency and the second counter ticks with 1 MHz for clock frequency independent time measurements. The counters are 64-bit values and readout of the lower 32 bits also latches the upper 32 bits. Table 5 shows the I/O devices and the registers.

Bit	Status		Control	
0	TRE	TX Transmit ready	–	–
1	DAV	RX Data available	–	–

Table 6: UART status bits

Timer The timer device provides a means to measure time as well as to trigger an interrupt at a certain point in time. It provides two 64-bit counters. While the first counter is incremented every clock cycle, the second counter is incremented every microsecond.

Interrupts can be triggered by storing a value in the "cycle interrupt time" and "μs interrupt time" registers. The timer device will then trigger an interrupt when the respective counter reaches the value provided in that register. The "cycle" interrupt is tied to interrupt 0; the "μs" interrupt is tied to interrupt 1.

To read out the 64-bit counter values consistently, the low word (at the higher address) must be read first. This latches the high word of the counter into an internal register, which is then returned when reading the high word (at the lower address). Similarly, the low word of the interrupt times must be written first. The write to the internal 64-bit register takes effect when the high word is written.

UART The UART is a minimal IO device for `stdout` and `stdin`. It is also used for program download. Table 6 shows the bits of the control register.

The UART address for `pasim` is defined in `patmos/simulator/include/uart.h`. For the compiler/library the constant is in `llvm/tools/clang/lib/Driver/Tools.cpp`.

3 Integration

Figure 10 shows the T-CREST platform. Several Patmos cores are connected via a memory tree [9] to a real-time memory controller [2, 19, 10] to the shared, external SDRAM memory. For efficient core-to-core communication each processor is connected to the Argo time-predictable network-on-chip (NoC) [27, 18, 16, 31].

These on-chip communication channels reduce the pressure on the shared memory bandwidth. Most of the T-CREST hardware is open-source under the industry friendly, simplified BSD license.[3] The open-source components are described in deliverable D 8.5, Open Source Reference Implementation.

3.1 Compiler and WCET Analysis

Besides the hardware, a compiler infrastructure has been developed in the project. The compiler for Patmos is an adaption of the LLVM compiler infrastructure [23, 13]. The time-predictable processor Patmos, with the available timing model, allows the development of WCET-aware optimization methods.

[3] see `https://github.com/t-crest`

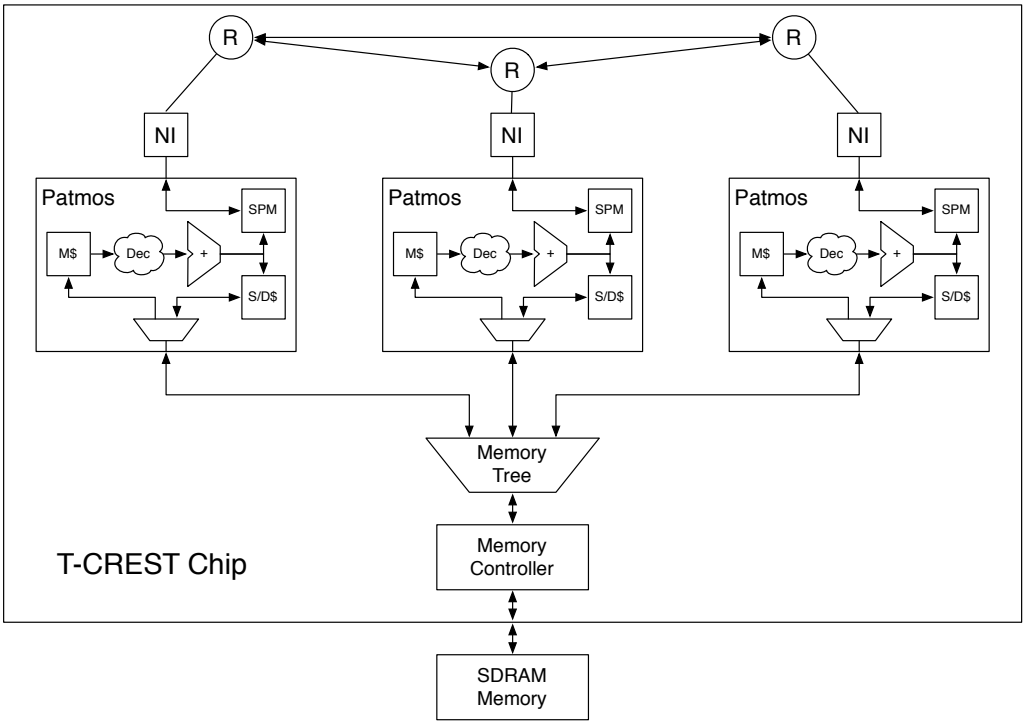


Figure 10: The T-CREST platform consisting of processor nodes that are connected via a NoC for message passing and a memory tree for shared memory access.

Exposing the micro-architecture at the ISA level and requiring the application to manage local memories leads to a small and fast hardware design and simplifies the WCET analysis since changes to the hardware state can be explicitly observed at the ISA level. This off-loads the tasks of using the available resources efficiently to the application code and thus to the compiler. While the compiler has a static view of the executed application, it also has a more high-level view than the processor. Together with less stringent resource and runtime requirements, since compilation is done at design time, the compiler can thus use more powerful optimizations.

3.1.1 Instruction Scheduling and If-Conversion

The VLIW architecture of Patmos requires the compiler to schedule instructions without hazards and to allocate instructions to the second pipeline. In our compiler, we use a standard bottom-up instruction scheduler to perform this task. Scheduling is performed after register allocation after register spill code and function prologues and epilogues have been inserted. In order to break false dependencies between instructions, the compiler can use register renaming during scheduling. Depending on the number of instructions available to fill control flow delay slots, the compiler decides to use either the delayed or non-delayed control flow instruction variants that are available in the Patmos ISA in order to reduce the number of No-Ops in the code.

The compiler makes use of the fully-predicated ISA to perform if-conversion, i.e., small conditionally executed basic blocks are converted into predicated straight-line code to avoid the overhead of branches. Our compiler can also eliminate all input-data dependent branches to generate single-path

code. For single-path code, all loops with input-data dependent loop iterations are converted into loops with predicated loop bodies and input-data independent loop counters, making the execution of the code and thus the execution time of the code independent of the actual values of the inputs. An input-data dependency analysis together with coding policies for single-path code enable the compiler to automatically determine which control-flow decisions depend on input data.

3.1.2 Support for the Stack Cache and Method Cache

The stack cache provides a fast way to store temporary data without necessarily writing the data back to global memory. The compiler therefore automatically uses the stack cache for spilling registers and for stack-allocated data. Stack cache management instructions are inserted in the prologue and epilogue of functions, as well as after all call sites to reserve and free stack frames in the stack cache [1]. However, due to the typed loads of Patmos, pointers to data allocated to the stack cache cannot be passed to callees. Therefore the compiler maintains a separate shadow stack in global memory, which is used for dynamically allocated stack data and for data which can escape the function where it is defined.

We have added support for the stack cache to the LLVM compiler, which supports the Patmos instruction set. First, loads and stores that shall end up in the stack cache use (stack) typed load and store instructions. Second, the compiler emits the `reserve`, `free`, and `ensure` instructions.

Local variables and data structures that are not accessed by a pointer cannot leak out of a function and can therefore be safely allocated on the stack cache. The compiler must also be able to figure out the maximum number of words used in a function. Therefore, memory allocated with `alloca()` with a non-constant size cannot be allocated on the stack cache. To fully support all legal operations on function local data of the C language, the compiler uses a second stack, the so-called *shadow* stack, for data which cannot be allocated in the stack cache.

As part of the function entry code, the compiler emits a `reserve` instruction to prepare the stack cache for the following load and store instructions. As part of the exit code, the compiler emits a `free` instruction to return the space on the stack cache. A `ensure` instruction after a call restores the call frame of the caller. Note that most ensure operations can be eliminated by the compiler in practice.

It has to be noted that the usage of the stack manipulation instructions around function call and returns is only one way to use the stack cache. Any well-formed part of the program can serve as a region for stack cache manipulation. These scopes can be several regions within a function or it can be a cross-function scope.

The method cache requires the compiler to split large functions into smaller sub-functions that fit into the cache. The compiler thus contains a function splitter pass that partitions functions into sub-functions of a predefined size. This not only enables the processor to execute arbitrarily large functions, but also reduces the costs of caching large functions of which only a fraction of the code is actually executed.

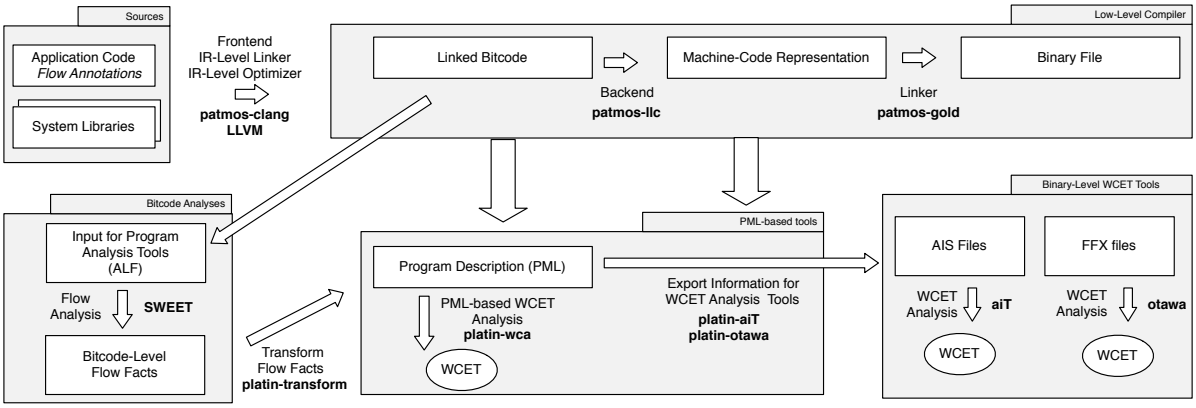


Figure 11: The T-CREST approach to compilation and WCET analysis.

3.1.3 Compilation and WCET Analysis

Finally, the compiler supports the WCET analysis by supplying meta-information about the program that is available in the compiler but lost in the final binary to the analysis, such as targets of indirect branches or possible addresses accessed by memory instructions. The compiler also uses feedback from the WCET analysis for optimization. For example, memory accesses through the data cache for which the value analysis of the WCET analyzer cannot determine the accessed address are replaced by accesses directly to global memory, so that such accesses do not introduce imprecision in the cache analysis.

Static WCET analysis has been an integral part of the Patmos toolchain since its early development stages. The WCET analysis tool aiT [12] from AbsInt has been adapted to support Patmos. In addition to precise WCET bounds, an important goal was to reuse existing platform-independent program analyses in order to benefit from advances in the rapidly evolving static analysis field. To achieve this goal, all information necessary for WCET analysis should be provided by the compiler and by analysis tools that operate on the platform-independent bitcode representation of LLVM. In this approach, a major challenge is the transformation and combination of compiler and analysis information, which is the main purpose of the `platin` tool that was developed for Patmos [23].

The `platin` tool supports the transformation of platform-independent flow information to machine code, using an approach that ensures sound results [13]. Furthermore, it prepares relevant analysis information for external binary-level WCET analysis tools, in particular the well-known industrial aiT tool [12]. The `platin` tool communicates with the compiler using PML files, which store all informations about the analyzed program that are relevant for WCET analysis. A distinguished feature of PML is that is allows to store information about both the platform-independent intermediate representation, and platform-dependent machine code. Figure 11 illustrates the integration of the compiler, WCET analysis, and external tools into our framework.

Additionally, `platin` provides a WCET analysis tool on its own, which takes advantage of properties that are characteristic for the Patmos architecture. First, our static analysis operates (almost) exclusively on information that was provided by the compiler and platform-independent analyses.

We therefore avoid to duplicate efforts of the compiler and do not need to model the semantics of machine code in detail.

Second, the Patmos design ensures that the timing of hardware components can be analyzed independently. This allows us to decouple different cache analyses as well as the pipeline timing analysis and to use global cache analyses (e.g., persistence analyses) that avoid costly virtual unpeeling.

Third, the Patmos design is meant to avoid the need for excessive context sensitivity for hardware timing analysis, in order to avoid scalability problems and allow modular analyses. For example, stack accesses cached by a conventional data cache do not pose a problem as long as the value of the stack pointer is known, but are unpredictable if full virtual inlining is intractable. In contrast, our analysis of the stack cache [15] does not require full context-sensitivity.

The `platin` tool, including the WCET analysis, is easy to adapt for different research experiments, and deployed as part of the open-source Patmos compiler.

3.2 Network-on-Chip

In order to build a chip-multiprocessor system out of Patmos processor cores we need a suitable interconnect – a network-on-chip (NoC). The Patmos multi-processor platform uses distributed, local memories connected to each processor. The NoC supports time-predictable data movement between these local memories.

To enable time-predictable usage of a shared resource the resource arbitration has to be time-predictable. In the case of a NoC, statically scheduled time-division multiplexing (TDM) is a time-predictable solution [27]. This static schedule is repeated and the length of the schedule is called the *period*. Like tasks in real-time systems, also the communication is organized in periods. The T-CREST NoC uses TDM from end to end, including the network interface. That approach also results in an efficient implementation of the network interface [31]. The latest T-CREST NoC uses a globally-asynchronous locally-synchronous style with asynchronous routers [18].

The Patmos is connected to the NoC via a network interface (NI). The NI has a standard OCP interface to the control registers and provides a read/write port to the local memory. Data is moved via the DMA in the NI between local memories of processors. The details of this interface and a library to support message passing can be found in deliverable D 3.8, Integration report of the full system implemented in an FPGA.

3.3 Memory Arbitration and Memory Controller

Even for embedded systems the on-chip available memory is usually too small to hold all code and data. Therefore, an off-chip SDRAM serves as shared main memory for the multicore processor. Access time to the SDRAM depends on the history of former accesses (e.g., open rows). This optimization improves the average case execution time, but not the WCET. Therefore, within T-CREST, memory controllers have been developed that are a better fit for real-time systems [2, 19, 10] This memory is connected by a second NoC, the so-called Bluetree [9].

The combination of the time-predictable memory NoC [9, 28] and the time-predictable memory controller allows, even on a CMP system, to provide upper bounds on memory transactions. This upper bound enables WCET analysis of individual tasks executing on a CMP system.

Several Patmos processors share the main memory via a memory arbiter. Tree versions of the arbiter are available: the Bluetree memory tree [9], a simple pseudo round-robin memory arbiter, and a distributed TDM arbiter [28]. The later arbiter is *pseudo* round-robin as it does not decide on all masters in the same cycle which to grant access, as this combinational decision does not scale. Instead, the arbiter sequentially 'polls' each master for access. Therefore, for each master that does not need access to main memory one clock cycle is 'wasted'. However, with enough cores the memory bandwidth is fully utilized and all cores are basically memory bounded.

For external memory we have implemented two different memory controllers for SRAM and SSRAM static memories and a time-predictable SDRAM controller [19].

3.4 Operating Systems for Patmos

Although not directly covered by the T-CREST project, two real-time operating systems (RTOS) have been ported to Patmos: RTEMS and TiCOS. The Real-Time Executive for Multiprocessor Systems (RTEMS) is an open source RTOS and popular in the avionics domain. The port of the well-known RTEMS is available at `https://github.com/t-crest/rtems`. The time-composable operating system (TiCOS) [4] is based on the open-source RTOS POK [7]. TiCOS and POK implement a two-level partitioned scheduler and provide an API according to the ARINC653 standard. TiCOS was ported by to Patmos [32] and is available at `https://github.com/t-crest/ospat`.

4 Evaluation

Besides being time-predictable, two properties of a processor are interesting: (1) its size and (2) its performance. We present first results of Patmos from an implementation in a low-cost FPGA.

4.1 Resource Consumption

Patmos is highly configurable with respect to the resource consumption. In this section we show synthesize results for Patmos in three different configurations: (1) standard configuration, (2) single-issue, and (3) a minimal version.

All results are from synthesizing Patmos for an Altera Cyclone II FPGA (EP2C70F896C6) and with a memory interface to the 32-bit SSRAM on the Altera DE2-70 FPGA board. The standard configuration of Patmos is dual-issue execution, a method cache of 4 KB with maximum 16 methods, a direct-mapped data cache of 2 KB, an instruction SPM of 8 KB, a data SPM of 2 KB, a boot SPM of 2 KB, and a memory controller for the external synchronous SRAM.

Component	Resources (LC)	Memory (KB)
Fetch	532	8
Decode	1,523	0
Execute	6,526	0
Memory	485	0
IO	288	2
Boot memory	13	2
Data cache	649	2.5
Method cache	1,921	4
SRAM controller	178	0
Total	11,878	18.5

Table 7: Resource consumption of Patmos components in the standard configuration

Configuration	Resources (LC)	Memory (KB)	fmax (MHz)
Standard	11,878	18.5	80.5
Single issue	6,111	18.5	82.1
Minimal	4,888	9.5	84.8

Table 8: Resource consumption and maximum clock frequency of different Patmos configurations

Table 7 shows the resource consumption of the individual components in the default configuration. We can see four of the 5 stages as dedicated components: Fetch, Decode, Execute, and Memory. Write back is just the write port of the register file and therefore not visible as a hardware component.

The decode stage contains the register file. As this register file is build out of LCs for the dual-issue version of Patmos, the resource consumption is quite high. Related to the dual-issue configuration is the size of the execution stage as it contains the full forwarding from the memory and write back stages of both pipelines to both execution stages. Another component that contributes significantly to the resource consumption is the method cache.

Table 8 compares the synthesize results for the three configurations of Patmos. We see that the single-issue version of Patmos reduces the footprint by almost 50%. The register file can now be implemented in on-chip memory, only a single ALU is needed, and the biggest resource saving comes from the simpler forwarding network.

For the minimal configuration of Patmos we have reduced all caches and SPMs to 1 KB. This reduces the amount of on-chip memory usage. In the minimal configuration the method cache is also restricted to cache only two methods. Therefore, the number of LCs is also reduced as the tag memory for the method cache is implemented in LCs.

Processor	CoreMark/MHz
Patmos, dual-issue	1.90
Patmos, single-issue	1.78
LEON3	1.96
MicroBlaze	1.90
NIOS II	1.49

Table 9: CoreMark scores for Patmos, LEON3, MicroBlaze, and NIOS II

4.2 Average-Case Performance

To evaluate the average-case performance of Patmos, we use the CoreMark benchmark[4] and compare the results to other FPGA-based processors. In particular, we compare Patmos to the Aeroflex Gaisler LEON3, Xilinx MicroBlaze, and Altera NIOS II processors.

The available CoreMark scores for these processors were obtained with 16 KB instruction cache and 16 KB data cache. We use a comparable configuration of Patmos for this evaluation, with a 16 KB method cache that can hold 16 methods, and a 16 KB data cache. The stack cache and the SPMs are disabled. With such a setup, the CoreMark benchmark fits into the caches, such that the benchmark evaluates the processor pipeline rather than the efficiency of the memory subsystem.

The results for Patmos, LEON3, NIOS II, and MicroBlaze were obtained on different FPGAs. Therefore, the operation frequency and the absolute CoreMark scores are incomparable. However, we can use the CoreMark/MHz measure to evaluate the efficiency of the instruction set and the compiler.

For the dual-issue version, Patmos achieves 1.90 CoreMark/MHz, while for the single-issue version it achieves 1.78 CoreMark/MHz. The score for LEON3 is 1.96 CoreMark/MHz and for MicroBlaze (version v8.20.b) it is 1.90 CoreMark/MHz. The NIOS II variant with 16 KB instruction cache and 16 KB data cache achieves 1.49 CoreMark/MHz. These scores are subsumed in Table 9.

We conclude that the performance of the Patmos pipeline is in the same range as for comparable processors that are not optimized for time predictability. However, the results also indicate that there may be room for improvement in the Patmos ISA and the compiler. The speed-up of the dual-issue version over the single-issue version is merely 6.7%. Further work on the compiler instruction scheduler is necessary to use the second pipeline more efficiently.

4.3 Method Cache

We evaluate the proposed method cache with embedded benchmarks from the Mälardalen benchmark suite [11] and from the MiBench benchmark suite, with different hardware settings, and with variations in the compiler settings. The Patmos processor was configured as follows: (1) 80 MHz clock frequency, (2) dual issue pipeline, (3) 4 KB direct-mapped data cache, (4) two 2 KB data and instruction scratch pad memories, (5) a small boot ROM, and (6) the method cache with varying sizes and function counts.

[4] see `http://www.eembc.org/coremark/`

Table 10: Hit rate for the `fft1` benchmark in different cache variations and configurations

Size	Functions	Hit rate (V)	Hit rate (F)
1 KB	8	13.58 %	12.47 %
1 KB	16	13.58 %	13.54 %
1 KB	32	13.58 %	13.54 %
4 KB	8	18.89 %	18.89 %
4 KB	16	37.59 %	35.49 %
4 KB	32	47.57 %	37.90 %
8 KB	8	18.89 %	18.89 %
8 KB	16	37.59 %	35.98 %
8 KB	32	83.71 %	53.15 %
16 KB	8	18.89 %	18.89 %
16 KB	16	37.59 %	37.59 %
16 KB	32	83.71 %	73.19 %
32 KB	8	18.89 %	18.89 %
32 KB	16	37.59 %	37.59 %
32 KB	32	83.71 %	83.71 %

The target board is the Altera DE2-70 development board, which contains a low-cost Altera Cyclone II EP2C70 FPGA. The external main memory is a synchronous SRAM of 2 MB with a 32-bit data bus. The memory controller accesses the external memory with bursts of four 32-bit words. The latency for the first read is three clock cycles. Therefore, reading 16 bytes takes seven clock cycles.

Our compiler performs function inlining, which not only has the advantage of removing call overheads and enabling further optimization opportunities, it can also eliminate small functions which would otherwise occupy space in the tag memory and thus cause the cache to evict functions, even if the method cache size is not full.

The Patmos compiler also splits functions into smaller sub-functions. The first task of this special compiler optimisation is to ensure that all sub-functions fit into the cache. The second task is to reduce the amount of code that is loaded into the cache but not executed, which is done by extracting some of the conditionally executed code into separate sub-functions that are only loaded into the cache when needed.

The current function splitter implements a simple heuristic to optimize the sub-functions. In future work we will improve the function splitter to also take the maximum number of functions that can be in the method cache into account.

For comparing individual features we picked the `fft1` benchmark. This benchmark performs forward and backward FFT in floating point. As Patmos does not contain a floating point unit, this benchmark uses the software floating point library. The `fft1` benchmark consists of 16 functions (including floating point library functions) that are executed, whose sizes vary between 72 bytes and 2.2 KB, amounting to a total of 9.5 KB of code. The default compiler setup splits the program into 53 sub-functions of an average size below 256 bytes.

Table 11: Hit rate for the `fft1` benchmark in different cache configurations with preferred function size of 1 KB

Size	Functions	Hit rate	Stall cycles	Utilization	Max. m.
4 K	8	40.07 %	201775	47.72 %	8
4 K	16	41.92 %	192920	49.77 %	13
4 K	32	41.92 %	192920	49.77 %	13
8 K	8	40.42 %	199969	48.13 %	8
8 K	16	53.18 %	143584	55.69 %	16
8 K	32	98.50 %	3738	74.54 %	25
16 K	8	40.42 %	199969	48.13 %	8
16 K	16	53.18 %	143584	55.69 %	16
16 K	32	98.50 %	3738	74.54 %	27
32 K	8	40.42 %	199969	48.13 %	8
32 K	16	53.18 %	143584	55.69 %	16
32 K	32	98.50 %	3738	74.54 %	27

4.3.1 Variable-Size versus Fixed-Block Method Cache

Table 10 compares the variable-size method cache (V) with the fixed-block method cache (F) for different sizes and different maximum numbers of functions. The compiler was set to split the functions into a maximum of 256 bytes. The table shows hit rates for the caches.

With a 1 KB method cache, the limit is the method cache size, as the hit rate does not change with the number of maximum functions. However, from 8 KB up to 32 KB the hit rate does not change with the cache size, but with the number of allowed functions. In this case the hit rate is limited by the maximum number of functions that can be cached. For 8 KB and 16 KB configurations, we can notice a significant difference between the variable-size and the fixed-block method cache. The variable-size method cache provides a higher hit rate than the fixed-block method cache.

4.3.2 Compiler Settings

Table 11 and Table 12 show the impact of function-splitting for different cache configurations for the variable-size method cache. Besides the hit numbers, the tables show the number of method cache stall cycles, the utilization of the loaded code, and the maximum number of functions that are fully stored at the same time in the cache. The utilization is the ratio of executed instructions and the number of instructions in a function, i.e., a higher utilization means more of the loaded instructions are actually executed.

Table 11 shows the results with a compiler setting that splits code into sub-functions of up to 1 KB. Table 12 shows the results for the default setting of splitting into 256 byte sub-functions. With bigger sub-functions the hit rate increases since the application is split into only 27 sub-functions instead of 48 sub-functions as in the default setup, and larger sub-functions are more likely to be reused. Furthermore, cache configurations with up to 32 functions and at least 8 KB of cache size can cache

Table 12: Hit rate for the `fft1` benchmark in different cache configurations with preferred function size of 256 B

Size	Functions	Hit rate	Stall cycles	Utilization	Max. m.
4 K	8	18.81 %	156933	74.34 %	8
4 K	16	37.55 %	122150	81.15 %	16
4 K	32	47.57 %	102620	82.97 %	30
8 K	8	18.81 %	156933	74.34 %	8
8 K	16	37.55 %	122150	81.15 %	16
8 K	32	83.84 %	30646	84.96 %	32
16 K	8	18.81 %	156933	74.34 %	8
16 K	16	37.55 %	122150	81.15 %	16
16 K	32	83.84 %	30646	84.96 %	32
32 K	8	18.81 %	156933	74.34 %	8
32 K	16	37.55 %	122150	81.15 %	16
32 K	32	83.84 %	30646	84.96 %	32

all required sub-functions simultaneously in the cache, i.e., each sub-function is only missed once and we observe a huge drop in the number of stall cycles.

In Table 12 we observe lower hit rates for a larger number of smaller sub-functions. However, the stall cycles, and thus the performance of the cache, improves in some cases over the previous setup. Smaller sub-functions are less likely to contain code that is loaded into the cache but not executed, which is also reflected by a higher utilization rate. The downside of smaller sub-functions is that even with 32 functions, cache entries are evicted from the cache even if there is space left in the method cache, as 32 sub-functions of an average size of less than 256 bytes take up at most 8 KB of cache. We can observe this in Table 12, where for 32 functions, increasing the cache size beyond 8 KB has no effect, and for smaller tag memories doubling the 4 KB cache has no effect.

4.3.3 Multiple Benchmarks

In this subsection we use one method cache configuration (8 KB and 32 functions) and evaluate the hit rate for all benchmarks from the Mälardalen [11] and the MiBench benchmark suites.

Figure 12 shows a scatter plot of the hit rate over the code size of the benchmark. We see a great variation of the code sizes with more small benchmarks than large benchmarks. We observe no correlation between the code size and the hit ratio. Therefore, the hit rate of the method cache is independent of the code size.

Figure 13 shows a bar graph with the hit rate for each evaluated benchmark.[5] For the benchmarks close to 100% hit rate, all functions fit into the cache and therefore each function has to be loaded only once. The benchmarks where the hit rate is low are small applications with a single main routine and no further functions. Therefore, the cache loads this main function on startup and the hit rate is

[5]The benchmarks denoted by *-tiny* are modified setups with decreased loop counts or smaller input files to keep the runtime on the simulator low.

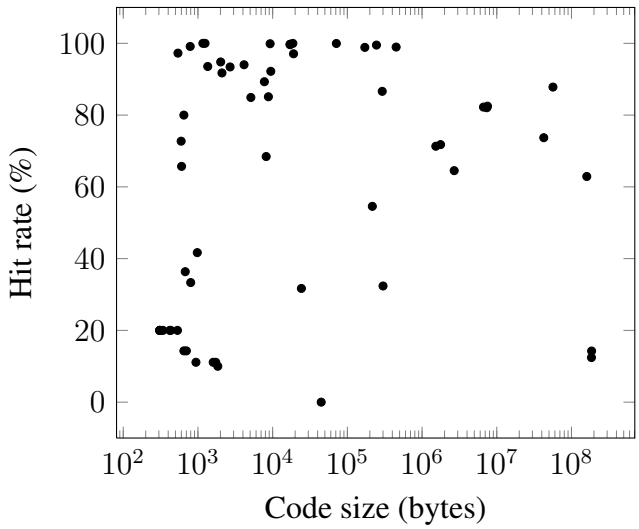


Figure 12: Hit rate over code size for 59 different benchmarks for a 8 KB method cache with maximum 32 methods

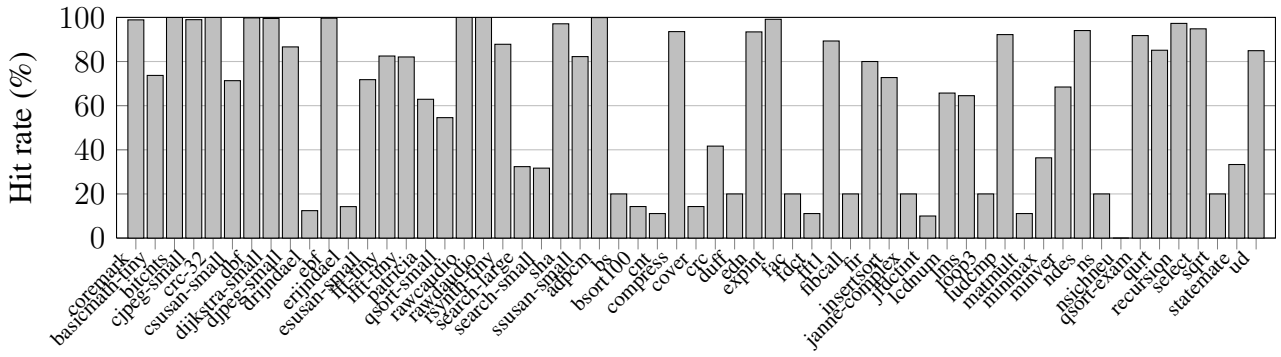


Figure 13: Hit rate for 59 different benchmarks for a 8KB method cache with maximum 32 methods

not significant for the execution since no replacement inside the cache is executed. The rest of the benchmarks achieve a good hit rate that is above 50%.

4.4 Stack Cache

We have implemented the stack cache in the Patmos processor in hardware and in the cycle accurate software simulation. In the evaluation section we report on the hardware size, compare with a standard caches. With the software simulation of Patmos we collect runtime statistics on stack and data cache usage with embedded benchmarks.

	Cache Size (KB)	Line Size (Words)		
		2	4	8
Direct Mapped	1	1.4	1.2	1.1
	2	2.7	2.3	2.2
	4	5.3	4.7	4.3

Table 13: Total cache size in KB for different line sizes and cache sizes.

4.4.1 Hardware Resource Consumption

The Patmos processor and the stack cache are implemented in an FPGA. For the evaluation we use the Altera Cyclone II FPGA on the DE2-70 FPGA board. Without the stack cache, Patmos consumes about 2400 logic cells and can be clocked at 77 MHz. Adding the stack cache increases the total size of Patmos to about 3200 logic cells and reduces the maximum clock frequency to 73 MHz. We consider the additional 800 logic cells on the high side for this cache implementation. We intend to optimize the code for size and pipeline speed.

The size of the stack cache is configurable and for first spill and fill tests the default configuration is 64 32-bit words. Therefore, it consumes a single on-chip memory block.

The main differences between a normal data cache and the stack cache are:

- A normal data cache needs hit detection by comparing the tag memory of each way with part of the address. This hit detection is usually on the critical path. In can be performed in parallel on data read, but needs to performed before a write, which might add an additional cycle for a store instruction.
  With the stack cache we have guaranteed hits on load and store instructions and no need to compare with a tag memory. The equivalence to hit detection in the stack cache happens on `reserve` and `ensure`, which happens less often than loads and stores.
- A normal data cache needs a tag memory, which can consume a considerable amount of memory. In the stack cache only two pointers into the address space mark which data is in the cache and which data is only in the main memory.

In a standard cache, each cache line contains a tag word and a valid bit. The size of the tag word depends on the cache size, the cache line length, and the associativity. Table 13 shows the total memory size (tag and data memory) for different configurations of a direct mapped cache. We can see that the tag memory adds up to 40% of memory consumption to the data cache. In contrast our stack cache needs only the two registers to mark the address range that is in the cache. The two pointers also serve for the valid bit. Therefore, the total size of the stack cache is equal to size of the data memory used for a cache.

4.4.2 Stack Cache Performance

The intention of the stack cache is to simplify WCET analysis by splitting different data areas to specialized data caches. Within the T-CREST project the WCET analysis tool aiT form AbsInt will be adapted to support the stack cache and other features of Patmos.

In the mean time we can evaluate the stack cache by average-case measurements. The T-CREST project also contains a cycle accurate software simulator of Patmos [8]. Therefore, it is possible to collect usage data and hit/miss rates for different stack cache configurations and compare them with a standard data cache.

In order to evaluate the stack cache we compiled and executed a subset of the publicly available benchmark suite MiBench on the Patmos simulator. These benchmarks cover a representative set of tasks often encountered in embedded systems, e.g., in telecommunication and automotive domains.

Each benchmark was first compiled to LLVM bitcode by the `clang` C frontend using optimization level `-O3` and then linked and optimized using `llvm-ld`. LLVM's `llc` tool generated Patmos machine code using two configurations: (1) with stack cache support enabled and (2) with stack cache support disabled, i.e., all stack data is kept in the main memory. With stack cache disable, the normal data cache caches the stack allocated data. The `gold` linker finalizes the code layout using a default memory layout, where all code and data sections are placed into Patmos' main memory.

The Patmos simulator executes the benchmarks and is configured with a 1/4 KB stack cache, a 8 KB data cache, and a 64 KB instruction cache, organized as method cache [24]. A larger stack cache of 1 KB is big enough to cache stack data without the need to spill stack frames to main memory. Therefore, we reduced the size of the stack cache to observe some spill and fill operations.

The stack cache is organized in word-sized blocks (4B), while the data and method caches are organized in 32-byte blocks. The 4-way set-associative data cache uses a least-recently-used (LRU) replacement policy and a write-through strategy with no-write allocation. The method cache likewise uses an LRU replacement policy. Transferring a cache block (32B) to or from main memory is assumed to take 40 cycles.

MiBench offers a small and a large data set for most benchmarks. We run most of the benchmarks with the default data set. For some benchmarks we use smaller data sets. This is indicated by the suffix in the figures.

4.4.3 Runtime

In our first experiment we compare the total number of execution cycles for each benchmark with stack cache support enabled against a variant with the stack cache disabled. Enabling the stack cache allows us to (partially) allocate the stack frame of functions to the stack cache instead of the main memory. This reduces the number of accesses through the data cache, but at the same time increases the number of instructions, because dedicated instructions, emitted by the compiler, manage the the stack cache explicitly. As shown by Figure 14, enabling the stack cache reduces the total number of execution cycles for certain benchmarks.

The gains are explained by (a) the additional cache space in the stack cache and (b) the handling of writes by the data cache. The additional cache space (1/4 KB) in the stack cache reduces the number of loads from the data cache and thus the number of accesses to the slow main memory. Additional gains are due to the long latency of stores to the data cache, which uses a write-through strategy with no-write allocation.

Some benchmarks profit less from the stack cache, most notably `crc-32`, `rawcaudio`, and `rawdaudio`. Loops that do not contain any spill code or function calls dominate these bench-

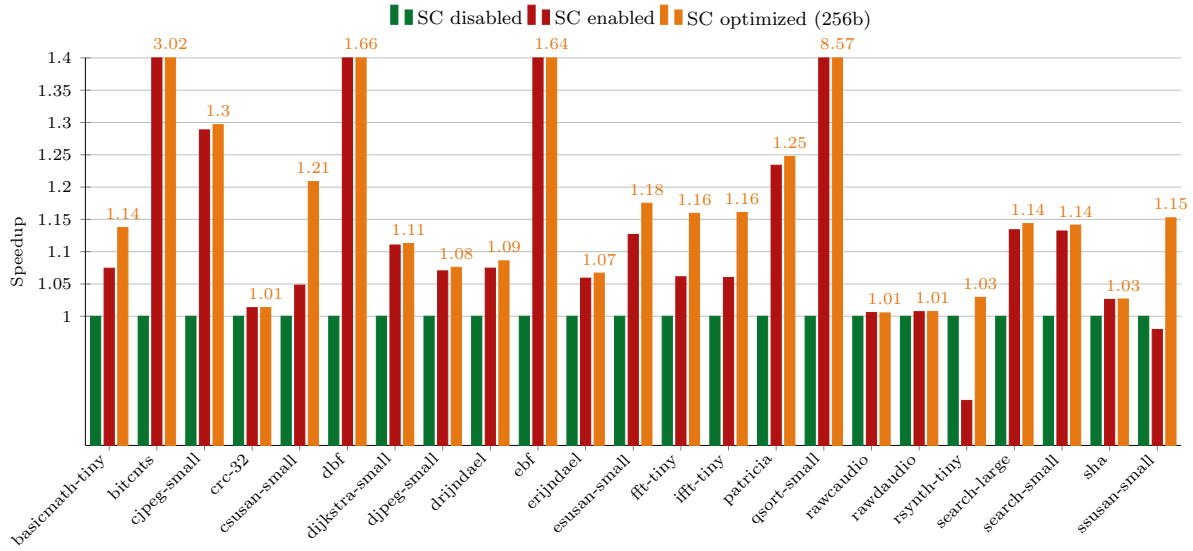


Figure 14: Speedup when running benchmarks with stack cache support disabled, enabled, and enabled with optimization (cache size 256 bytes, larger is better).

marks. The simple stack cache allocation strategy thus cannot find any data to allocate to the stack cache.

We also evaluated the benefits of a simple compiler optimization that removes useless ensure operations (see SC optimized in Figure 14). An ensure can be eliminated after a call whose worst-case stack cache occupancy combined with the ensure's size does not exceed the stack cache size.

4.4.4 Stack Cache Utilisation

As shown before, using the stack cache can be quite profitable. We thus present some additional data characterizing how the various benchmarks use the stack cache. Figure 15 shows the transfer volume to and from the data and stack cache. The benchmarks that showed the best runtime improvements make heavy use of the stack cache. For instance, 79% of the memory accesses of the `bitcnts` go to the stack cache. With regard to data transfer volume, these numbers even increase. While roughly 65% of the accesses of the `patricia` benchmark target the stack cache, almost 75% of the transfer volume is serviced by the stack cache.

Benchmarks with little or no gain rarely make use of the stack cache, as shown by Figure 14. The main reason is that these benchmarks are dominated by a single loop without any spill code or function calls. This is confirmed by the numbers in Figure 16, which shows the amount of dynamically allocated data on the stack cache.

Indeed, the current algorithm to allocate data on the stack cache only leverages compiler-generated spill slots, which are often linked to function calls (saving/restoring registers before and after calls). However, we expect that more powerful allocation algorithms will be able to overcome this limitation. For instance, the `rawcaudio` and `rawdaudio` benchmarks mostly operate on small buffers, which are potential candidates for stack cache allocation.

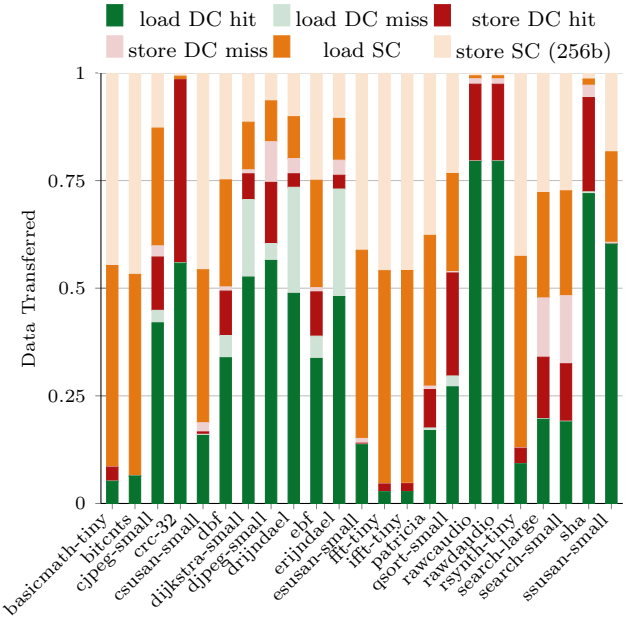


Figure 15: Normalized transfer volume of data accesses to the data and stack cache for each benchmark (data cache 8 KB, stack cache 256 bytes).

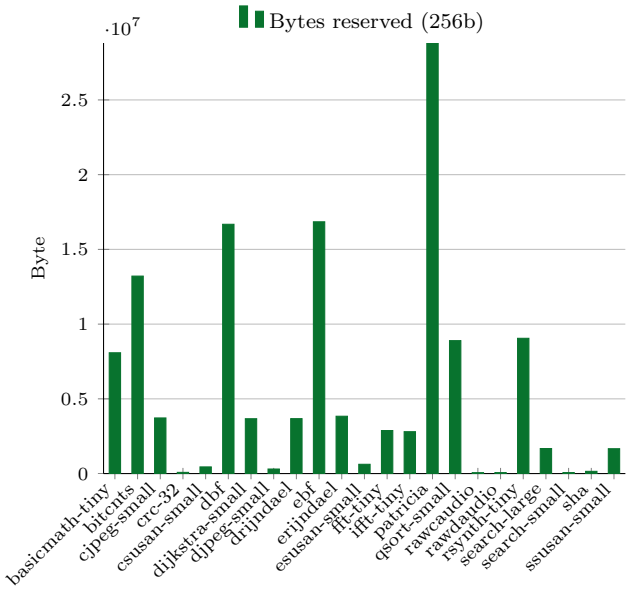


Figure 16: Data dynamically allocated on the stack cache. The stack cache usage correlates with the number of executed function calls (cache size 256 bytes).

5 Build Instructions

In the following we present the Patmos build instructions on a Linux/Ubuntu system (13.10).[6] Patmos and the compiler have also been successfully installed on a Mac OSX system. The support of Windows is marginal, or basically not existent.

5.1 Setup On Ubuntu 13.10

After a plain Ubuntu installation several packages need to be installed. The following apt-get lists the packages that need to be installed:[7]

```
sudo apt-get install git openjdk-7-jdk gitk cmake make g++ texinfo flex bis
  subversion libelf-dev graphviz libboost-dev libboost-program-options-dev
  ruby1.9.1-dev liblpsolve55-dev python zlib1g-dev gtkwave gtkterm scala
```

The following gem install might not be needed as with the correct packages (liblpsolve55-dev) the setup script shall do it automatically. Needs to be checked on a fresh Ubuntu machine.

From within `llvm/tools/platin` install:

```
sudo gem1.9.1 install lpsolve --pre
sudo gem1.9.1 install ext/lpsolve-5.5.10.j.gem
```

Install `sbt` with:

```
wget http://dl.bintray.com/sbt/debian/sbt-0.13.2.deb
sudo dpkg -i sbt-0.13.2.deb
sudo apt-get update
sudo apt-get install sbt
```

For the Quartus setup it is best to change the default shall to `/bin/bash`:

```
sudo rm /bin/sh
sudo ln -s /bin/bash /bin/sh
```

Ubuntu 12.04 LTS We also tested the setup on long-term support release 12.04 LTS. The compiler (gcc) is recent enough for all T-CREST code. Only some additional packages need to be installed:

```
sudo apt-get install python3-lxml
```

[6] I used the 32-bit version of Ubuntu to simplify the Quartus installation.
[7] Some packages might be available in newer version when reading this document.

5.1.1 Building Patmos and the Compiler Tool Chain

We assume that the T-CREST project will live in `$HOME/t-crest`. Patmos and the compiler can be checked out from GitHub and built as follows:

```
mkdir ~/t-crest
cd ~/t-crest
git clone https://github.com/t-crest/patmos-misc.git misc
./misc/build.sh
```

For developers with push permission generate an ssh key and upload it at GitHub (see `https://help.github.com/articles/generating-ssh-keys` for detailed instructions). The ssh based clone string for write access is then:

```
git clone git@github.com:t-crest/patmos-misc.git misc
./misc/build.sh
```

This script (`build.sh`) will checkout several other repositories (the compiler, library, the Patmos source, and benchmarks) and builds the compiler, the Patmos simulator, and the test benches. Therefore, take a cup of coffee and find some nice reading.

The `build.sh` script contains default options, which should work out of the box. The build settings can be changed by a customized `misc/build.cfg` file. The file `misc/build.cfg.dist` is an example configuration file containing default values. It is ignored by the build process and should not be edited. [8] To change any options for `misc/build.sh`, either start with an empty `misc/build.cfg` or copy `misc/build.cfg.dist` and modify the values to your need.

After building the compiler add the path to the compiler executables (e.g., into your `.bashrc` or `.profile`): [9]

```
export PATH=$PATH:$HOME/t-crest/local/bin
```

Optionally, you may additionally add the `misc` checkout to your path, so that `build.sh` and the helper tools in `misc` can be executed from everywhere.

```
export PATH=$PATH:$HOME/t-crest/misc
```

A complete logout from Ubuntu might be needed to take effect (just closing a terminal window is not enough, depending on how you set up your profile files). You can test your installation by checking if the compiler is available:

```
patmos-clang --version
```

For correct correct signing of your changes set the username and email in git with:

```
git config --global user.name "Joe Someone"
git config --global user.email "joe.someone@domain.com"
```

[8] It is autogenerated by `build.sh -e` from the values in `build.sh`.

[9] The path needs to be absolute. LLVM cannot handle a path relative to the home folder `~`, e.g., `~/t-crest/local/bin`.

5.1.2 Quartus

Download the free web edition of Quartus from Altera. The Linux version is installed as follows:[10]

```
tar xvf Quartus-web-xxx.tar
```

The software installation is started with a plain `setup.sh`.[11] Then add the bin directory of Quartus to your $PATH. For access to the serial port and access rights for the USB Blaster following additional steps are needed:

```
# Add user to dialout group for the serial port access
sudo usermod -a -G dialout user

# Add permissions to access the Altera USB Blaster
sudo su -
echo "SUBSYSTEM==\"usb\", DRIVER==\"usb\", ATTR{idVendor}==\"09fb\", \
ATTR{idProduct}==\"6001\", MODE=\"0666\"" > /etc/udev/rules.d/51-usbblaster

udevadm control --reload-rules
```

After fixing the permissions for the USB Blaster open Quartus and test if the cable is found with the programmer. Select USB-Blaster in *Hardware Setup*. When connected to an FPGA test the USB-Blaster with *Auto Detect* (With the DE2-115, a question about shared JTAG ID pops up – select EP4CE115).

Quartus 13.1 drops the support of Cyclone II devices. Therefore, the (phased out) Altera DE2-70 board is not supported anymore. Version 13.0 supports Cyclone II till Cyclone V devices and might be the best option at the moment.

5.2 Setup On Mac OS X

Several tools are needed, best installed with MacPorts. For Patmos simulator and assembler: `boost`, `libelf`. For simulation with ModelSim: `wine` For Aegean: python33, py33-lxml. Make a link from python3 to python3.3 as this is the way it is invoked.

5.3 Hello World

We can start with the standard, harmless looking Hello World:[12]

```
int main() {
    printf("Hello Patmos!\n");
}
```

[10] `http://www.altera.com/literature/manual/quartus_install.pdf`
[11] Or `bash setup.sh`
[12] This example code is not part of the distribution, but can be put at any directory.

With the compiler installed it can be compiled to a Patmos executable and run with the simulator as follows:

```
patmos-clang hello.c
pasim a.out
```

However, this innocent examples is quiet challenging for an embedded system: It needs a C compiler, an implementation of the standard C library, printf itself is a challenging function, the generated ELF file needs to be understood by a tool and the individual sections downloaded, and finally a terminal (often a serial line) needs to be available on the target, and your test PC needs to have a serial line as well and a terminal program needs to run.

Therefore, we might start from a minimal assembler program and execute that in the simulator and emulator. From that base we can build up too a multi-core version of Patmos that executes in an FPGA and bootstraps with programs loaded via a serial port.

5.4 Building Patmos

The whole build process of Patmos,[13] applications in assembler and in C, configuration of the FPGA, and downloading an application is `Makefile` based. The build of Patmos is within the `patmos` folder, therefore, the following descriptions assumes you have changed to:

```
t-crest/patmos
```

The complete design flow (including the LLVM based C compiler) can execute in a Linux machine. The flow without the C compiler should be able to execute in a Windows/Cygwin environment. Under Mac OS X all tools, except Quartus, are working (ModelSim under wine). For FPGA synthesis and configuration Windows XP within a VMWare virtual machine is a possible solution.

On a Linux box with the installed LLVM compiler and Quartus in your PATH, the complete build processes for a Hello World is as follows:

```
make BOOTAPP=bootable-bootloader APP=hello_puts \
   tools comp gen synth config download
```

However, this involves quite many steps. Therefore, we suggest doing some *manual* buildup to explore the full build process and the possibilities.

As a start we build some tools (e.g., the assembler, simulator, file conversion utility, and the boot loader). This has to be done once only.

```
make tools
```

[13] Get the source from GitHub with: `git clone git@github.com:t-crest/patmos`

5.4.1 A Few Assembler Instructions

We start with a very small assembler program that moves a few values into registers (see `asm/basic.s`). With following make command the program is assembled and executed in the software simulator of Patmos.

```
make swsim BOOTAPP=basic
```

The simulator options are set to write out the register contents after each instruction. The emulator (the Chisel based simulator) can execute the same program with following command:

```
make hwsim BOOTAPP=basic
```

This command assembles the application, executes the Chisel based hardware construction during which the program is used to initialize the on-chip ROM, generates a C++ based emulator, compiles that emulator, and executes it. The emulator shows the register content after each instruction.

Those two Patmos simulations, the software simulator and the Chisel based emulator, are used for a co-simulation based test. In this co-simulation all available assembler programs are executed in both simulations and the register out put is compared. The test can be started with:

```
make test
```

5.4.2 A C Based Blinking LED

As a first real example we build the embedded version of Hello World, the blinking LED, from a C program. You can find the C source in `c/blinking.c`.

```
make BOOTAPP=bootable-blinking comp gen synth config
```

Additionally to blinking an LED this program also writes alternating '0' and '1' to the serial port. Connect the FPGA board to your serial port, open a terminal of your choice (e.g., `gtkterm`), connect to the serial port, set the baud rate to 115200, no parity, and no handshaking. You should see alternating '0' and '1' sent out synchronous to the blinking.

Note that the program name (`blink`) is prefixed by `bootable-`. The marker selects the right compiler settings for a program that ends up in the Patmos on-chip ROM. As the on-chip memory is limited, only tiny programs are supported in this execution mode.

Figure 33 shows the code for the embedded Hello World C program. Two constants (0xF0000900 and 0xF0000804) are the addresses of the IO devices LED and serial port. IO devices connected to Patmos are connected to the local, uncached memory area. This is the same memory area where data SPM and NoC SPM are connected. Therefore, to access them one needs to use the local load/store instructions. With the attribute `_SPM` the compiler is instructed to emit the correct load and store instructions.

```
/*
    This is a minimal C program executed on the FPGA version of Patmos.
    An embedded Hello World program: a blinking LED.

    Additional to the blinking LED we write to the UART '0' and '1' (if available).

    Author: Martin Schoeberl
    Copyright: DTU, BSD License
*/

#include <machine/spm.h>

int main() {

    volatile _SPM int *led_ptr = (volatile _SPM int *) 0xF0000900;
    volatile _SPM int *uart_ptr = (volatile _SPM int *) 0xF0000804;
    int i, j;

    for (;;) {
        *uart_ptr = '1';
        for (i=2000; i!=0; --i)
            for (j=2000; j!=0; --j)
                *led_ptr = 1;


        *uart_ptr = '0';
        for (i=2000; i!=0; --i)
            for (j=2000; j!=0; --j)
                *led_ptr = 0;

    }
}
```

Listing 1: A blinking LED

5.4.3 Make Targets

A list of the most important make targets:

tools build of all tools, including the Patmos software simulator

asm assemble source (from folder `asm`)

swsim execute the Patmos simulator

hwsim execute the Patmos emulator

emulator build the Chisel based C++ emulator

comp compile a C program as loadable ELF binary

bootcomp compile a C program as a bootable image

gen generate the Verilog code

synth synthesize for an FPGA

config configure the FPGA

download download an elf file into the main memory via the Patmos bootloader

test run all assembler tests

The name of an application that can execute from the on-chip ROM is set with the BOOTAPP variable.

5.4.4 Download of ELF Files

On a Linux box with the installed LLVM compiler and Quartus in your PATH, the complete build processes for the Hello World is as follows:

```
make BOOTAPP=bootable-bootloader APP=hello_puts \
   tools comp gen synth config download
```

You should see the download information and then the greeting from Patmos:

```
/home/martin/t-crest/patmos/install/bin/patserdow -v /dev/ttyUSB0 /home/mar
Port opened: true
Params set: true
Elf version is '1':true
CPU type is:48875
Instruction width is 32 bits:true
Is Big Endian:true
File is of type exe:true
Entry point:131076
```

```
[++++++++++] 49778/49778 bytes
Hello, World!

EXIT 0
```

The `Makefile` use following variables to configure the build process: `BOOTAPP` is an application that ends in the on-chip ROM. This may be an assembler program or a simple C program; most prominent the boot loader for ELF binaries. A C program that shall be compiled as ROM target needs to be prefixed with `bootable-`. `APP` is a C program resulting in an ELF binary that can be either loaded by the emulator or the boot loader when executing in an FPGA.

Here an example of the individual steps to build the blinking LED C hello world (on a different FPGA board):

```
make tools
make BOOTAPP=bootable-echo bootcomp gen
make BOOTAPP=bootable-echo BOARD=bemicro synth
make BOARD=bemicro BLASTER_TYPE=Arrow-USB-Blaster config
```

This split of the make commands is for demonstration. It is possible to merge all steps into a single make (on Linux systems) or two steps when using two operating systems (e.g., Mac OSX for compilation and Windows for synthesis).

Emulator and elf File The emulator can read a standard ELF file. An example how to compile a small C program that uses part of the standard library and executing it on the emulator is as follows:

```
make emulator
make comp APP=hello_puts
install/bin/emulator tmp/hello_puts.elf
```

5.4.5 Supported FPGA Boards

At the time of this writing we have mainly focused on Altera FPGA based boards. Following boards are directly supported in the default build process:

- Altera DE2-70 (`altde2-70`)
- Altera DE2-115 (`altde2-115`)
- Altera/Farnell BeMicro (`bemicro`)

Setting the `BOARD` variable configures the board that shall be used. Without changing the `Makefile` the default of the board (and any other build variables) can be overridden by providing a local `config.mk` that is included in the `Makefile`.

5.4.6 Multicore Patmos

A multicore Patmos with shared external memory and the network-on-chip Argo is configured via the Aegean framework. Aegean is a collection of Python scripts that read in XML based configuration description (e.g., topology, network connections, processor types,...). Aegean generates the Chisel based components (Partmos, memory arbitration tree, and memory controller), generates the VHDL top-level to connect them with the VHDL based NoC, synthesizes the hardware, compiles the application for the individual cores, configures the FPGA, and downloads the application.

In directory `aegean` run:

```
make platform AEGEAN_PLATFORM=mandelbrot_demo
make synth config AEGEAN_PLATFORM=mandelbrot_demo
```

to generate (and synthesize) the mandelbrot application on a 4 core version of Patmos for an Altera DE2-115 FPGA board. This application is compiled into the on-chip memories and therefore executing right after configuration of the FPGA. Have a terminal open and connected to the serial port (115200 baud, 1 stop bit, no handshake) during and after the FPGA configuration and you shall see the output of the mandelbrot calculation.

However, the approach to have the application in on-chip memory works for tiny programs only. Furthermore, each software change needs a new synthesize run. A better approach is to build a platform that contains a bootloader (similar to the single core version) and some startup code to synchronize the program start with the other cores. This platform is the default configuration in Aegean and needs to be generated only once with:

```
make platform
make synth
```

The FPGA is configured from within the `aegean` directory with:

```
make config
```

The compilation and download of the application is then best done within the `patmos` directory with:

```
make APP=hello_puts comp download
```

This application is the same single core *Hello World* application that we used in Section 5.4.4. However, here we just compiled the application and downloaded it via the serial port. We synthesized and configured the FPGA from within the Aegean project for the multi-core version.

5.5 The Xilinx ML605 Platform

For the evaluation within the T-CREST project the Xilinx ML605 FPGA board was chosen as the 'standard' evaluation platform. The T-CREST platform contains, besides several Patmos' connected with the Argo NoC, a memory tree, called BlueTree, from UoY and the time-predictable memory controller from TU/e. As the building this platform needs some non-free tools and including closed source code, we provide prebuilt configurations of the T-CREST platform as bit files available for download:

- `http://patmos.compute.dtu.dk/`

To configure the FPGA use the IMPACT software from Xilinx (command `impact`), which is part of the Xilinx ISE package and needs no license (It is also available in a smaller Lab package). To compile an application and download it to the ML605 follow the exact same steps as for the Altera CMP version, e.g., from within the `patmos` directory:

```
make APP=hello_puts comp download
```

6 Requirements

In this section all requirements in aspect CORE and scope NEAR from Deliverable D 1.1, which are relevant for the processor work package, are listed. NON-CORE and FAR requirements are not repeated here. The requirements are followed by a comment to what extent it is fulfilled by the prototype or if it is not (yet) relevant to the processor development.

P-6-001 The processor (and local memory/cache) shall be a fully timing compositional architecture to support modular WCET analysis.

Patmos is fully timing compositional.

P-2-002 The processor shall support a standard RISC instruction set (similar to the MIPS ISA).

The instruction set of Patmos is a RISC instruction set with full predication.

P-2-003 The processor shall have two execution pipelines and a shared register file with full forwarding.

Patmos contains two execution pipelines. If resource consumption shall be reduced, Patmos can be configured for a single execution pipeline.

P-2-004 The processor shall support time-predictable (WCET analysable) caches.

Patmos includes a stack and method cache to simplify WCET analysis

P-2-005 The processor shall support single-path code with predicates.

All instructions are predicated.

P-2-006 The processor shall have a single network port for memory access.

All caches and the cache bypass load/store are multiplexed into a single OCP port.

P-0-061 The processor shall have a cycle counter, which can be read out for performance analysis.

Patmos contains two cycle counters: one ticking at the clock frequency and one ticking at 1 MHz.

P-4-014 The processor shall support time control instructions.

Patmos has a programmable timer interrupt.

P-4-015 The processor shall have variants of load/store instructions to support the different varieties of memory: main memory, stack and SPM.

Patmos supports typed load and store instructions for different data areas.

P-4-016 The processor shall provide access to the DMA controller via special block memory copy instructions.

The DMA controller of the network interface is configured with local load and store instructions.

P-4-019 The processor shall be time-predictable (i.e.: temporal bounds on the time to produce and consume outstanding requests, to execute configuration code and to transport the configuration settings to the DRAM controller shall be provided).

The architecture of Patmos is time-predictable.

P-5-021 The processor shall have a local memory from/to which data has to be moved explicitly to/from the (shared) main memory.

Patmos contains local memory for data and instructions.

P-5-022 The latency of instructions (except instructions to access main memory) shall be invariable with respect to their operands and statically known.

All instructions (except memory access) have a constant execution time.

P-5-023 The processor shall support full predication on its ISA.

All instructions are predicated.

P-5-025 The processor shall provide instructions to bypass the cache.

Patmos contains typed load and store instructions. One type is cache bypass to main memory.

P-5-065 The processor shall provide non-blocking memory operations.

Access to local memory is non-blocking. A cache line fill or cache bypass load is blocking the pipeline.

P-6-038 The processor shall implement disjoint instruction and data caches.

Patmos implements a method, a stack, and a data cache.

P-6-039 The processor shall use LRU replacement policy for both caches.

The data cache can be configured to a 2-way LRU implementation. The method cache implements a FIFO replacement.

P-6-042 Caches shall be private.

All caches are processor local. Cache coherence between the data caches can be enforced by cache flushing.

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.

Patmos can use interrupts (e.g., the time interrupt) for preemption.

P-0-506 Priority-preemptive scheduling The system shall provide means to implement CPU-local priority-preemptive scheduling without migration of threads between CPUs.

Patmos supports preemptive scheduling with the timer interrupt.

P-0-508 Interrupts: The CPU shall support interrupts.

Patmos supports interrupts.

P-0-509 Exceptions: The CPU shall support exceptions.

Patmos supports exceptions.

P-0-525 Cache: The architecture shall contain predictable instruction and data caches.

Patmos supports predictable cacheing for stack, data, and full functions.

P-0-526 Data cache flushing and bypassing: The platform shall provide a cache control interface that allows application code to flush and bypass the data cache.

All caches can be flushed. The data cache can be bypassed.

P-0-543 Data cache freezing: The platform should provide a cache control interface that allows application code to freeze the data cache.

Patmos does not support cache freezing.

P-0-527 Scratchpad: The platform shall contain data scratchpad memory with bounded access time.

Patmos contains a data scratchpad memory with single cycle access time (no pipeline stalling).

P-0-528 Scratchpad Control: The tool-chain shall provide a scratchpad control interface (e.g., code annotations) that enables managing data in scratchpads at design time.

This requirement is not applicable for Patmos. The compiler supports macros for scratchpad access.

P-0-538 Cache Bypass: The processor shall provide mechanisms to bypass the data cache on store and load instructions.

Patmos supports typed load and store instructions, where one type is cache bypass.

7 Conclusion

In this deliverable we presented the design of the time-predictable processor Patmos and its integration into the T-CREST platform. We believe that future embedded real-time systems need processors, interconnect, and memory controller that are designed to minimize the WCET and only implement

architectural features that are WCET analyzable. To provide good single thread performance Patmos implements a statically scheduled, dual-issue pipeline. Patmos serves as platform for future research on co-development of time-predictable architecture features and their WCET analysis beyond the T-CREST project.

References

[1] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. A time-predictable stack cache. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable sdram memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, New York, NY, USA, 2007. ACM.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[4] Andrea Baldovin, Enrico Mezzetti, and Tullio Vardanega. A time-composable operating system. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, volume 23 of *OASICS*, pages 69–80. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[5] Florian Brandner, Stefan Hepp, and Alexander Jordan. Criticality: static profiling for real-time programs. *Real-Time Systems*, pages 1–34, 2013.

[6] Philipp Degasperi, Stefan Hepp, Wolfgang Puffitsch, and Martin Schoeberl. A method cache for Patmos. In *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*, Reno, Nevada, USA, June 2014. IEEE.

[7] Julien Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. In *13th Real-Time Linux Workshop*, volume 10, 2011.

[8] DTU. D 2.1 software simulator of patmos. Technical report, T-CREST: `http://www.t-crest.org/page/results`, 2012.

[9] Jamie Garside and Neil C Audsley. Investigating shared memory tree prefetching within multimedia noc architectures. In *Memory Architecture and Organisation Workshop*, 2013.

[10] Manil Dev Gomony, Benny Akesson, and Kees Goossens. Architecture and optimal configuration of a real-time multi-channel memory controller. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1307–1312, 2013.

[11] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[12] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Online, last accessed November 2013].

[13] Benedikt Huber, Daniel Prokesch, and Peter Puschner. Combined WCET analysis of bitcode and machine code using control-flow relation graphs. In *Proceedings of the 14th ACM SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems (LCTES 2013)*, pages 163–172. The Association for Computing Machinery, 2013.

[14] Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object cache. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, pages 26–35, New York, NY, USA, 2010. ACM.

[15] Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems (RTNS 2013)*, pages 55–64, New York, NY, USA, 2013. ACM.

[16] Evangelia Kasapaki, Martin Schoeberl, Rasmus Bo Sørensen, Christian T. Müller, Kees Goossens, and Jens Sparsø. Argo: A Real-Time Network-on-Chip Architecture with an Efficient GALS Implementation. 2014. submitted.

[17] Evangelia Kasapaki and Jens Sparsø. Argo: A Time-Elastic Time-Division-Multiplexed NOC Using Asynchronous Routers. In *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 45–52, May 2014.

[18] Evangelia Kasapaki, Jens Sparsø, Rasmus Bo Sørensen, and Kees Goossens. Router designs for an asynchronous time-division-multiplexed network-on-chip. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 319–326. IEEE, 2013.

[19] Edgar Lakis and Martin Schoeberl. An SDRAM controller for real-time systems. In *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[20] Mälardalen Real-Time Research Center. WCET benchmarks. Available at `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, accessed 2009.

[21] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[22] Peter Puschner, Raimund Kirner, Benedikt Huber, and Daniel Prokesch. Compiling for time predictability. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 382–391. Springer Berlin / Heidelberg, 2012.

[23] Peter Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *9th Workshop on Software Technologies for Future Embedded and Ubiquitious Systems (SEUS 2013)*, pages 33–40, 2013.

[24] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[25] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, pages 11–16, Tokyo, Japan, March 2009. IEEE Computer Society.

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

[27] Martin Schoeberl, Florian Brandner, Jens Sparsø, and Evangelia Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*, pages 152–160, Lyngby, Denmark, May 2012. IEEE.

[28] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, 2014.

[29] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. Data cache organization for accurate timing analysis. *Real-Time Systems*, 49(1):1–28, 2013.

[30] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The Patmos approach. In *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)*, pages 11–20, Grenoble, France, March 2011.

[31] Jens Sparsø, Evangelia Kasapaki, and Martin Schoeberl. An area-efficient network interface for a TDM-based network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1044–1047, San Jose, CA, USA, 2013. EDA Consortium.

[32] Marco Ziccardi. A time-composable operating system for the Patmos processor. Master's thesis, Technical University of Denmark, 2013.