A Method Cache for Patmos

Philipp Degasperi
Institute of Computer Engineering
Vienna University of Technology
philipp.degasperi@rolmail.net

Stefan Hepp
Institute of Computer Languages
Vienna University of Technology
hepp@complang.tuwien.ac.at

Wolfgang Puffitsch, Martin Schoeberl
Department of Applied Mathematics
and Computer Science
Technical University of Denmark
wopu@dtu.dk, masca@dtu.dk

Abstract—**For real-time systems we need time-predictable processors. This paper presents a method cache as a time-predictable solution for instruction caching. The method cache caches whole methods (or functions) and simplifies worst-case execution time analysis. We have integrated the method cache in the time-predictable processor Patmos.**

We evaluate the method cache with a large set of embedded benchmarks. Most benchmarks show a good hit rate for a method cache size in the range between 4 and 16 KB.

I. Introduction

Embedded real-time systems are commonly used in computer applications in our daily life. In such systems, the execution time, and especially the worst-case execution time (WCET), are important. With our design we focus on the WCET such that timing constraints can be met and the system executes safely.

A performance issue in modern computer architectures is the increasing gap between processor speed and memory access latency, which is solved by introducing caches and building memory hierarchies. Traditional caches are optimized for the average case and are hardly time-predictable, which makes them unpractical for real-time systems. Therefore, we build time-predictable caches that are easy to integrate in modern WCET analyzers. A possible alternative to conventional instruction caches is the method cache. The idea of a method cache was first proposed by Schoeberl [15]. An implementation is currently used in two Java processors: in JOP [16] and in SHAP [21]. As Java uses the term "method", the cache is called "method" cache, but is also applicable for functions and procedures. In the remainder of this paper we talk about the method cache, but use functions that are cached, as our current processor is mainly programmed in C.

This paper presents the method cache design and its implementation in the time-predictable processor Patmos [20]. The method cache caches full functions and therefore cache misses can only occur on a function call or on a return from a function. All other instructions are guaranteed hits. We explore two variants of the method cache: (1) the fixed-block method cache and (2) the variable-size method cache. With the fixed-block method cache the cache is divided into blocks (like cache lines in a standard cache) and the allocation granularity of the cache for a function is in those blocks. The variable-size method cache is more flexible, as the allocation granularity is at word level. We find, by benchmarking the method cache with embedded benchmarks, that the variable-size method cache performs equal or better than the fixed-block method cache. However, this flexibility comes at the price of an increased hardware cost between 40 % and 65 %.

This method cache provides an important part of a time-predictable memory hierarchy, which is one of the research focuses of the T-CREST project. The work was carried out within the T-CREST project, which is a research project supported by the European Union in its FP7 funding program.

A. The T-CREST Project

The T-CREST[1] project aims at building a time-predictable multicore platform suited for hard real-time systems with a predictable and low WCET. Schoeberl has argued that time predictability is not quantifiable but that one can compute the WCET of a task and compare it on different architectures [17]. Being time-predictable is then an interaction between:

- the hardware architecture,
- the compiler, and
- the static WCET analysis tool.

The T-CREST platform takes this into account and proposes novel solutions for all these parts. The hardware consists of time-predictable processor cores (called Patmos) [20] that are connected through a statically scheduled time-division-multiplexed network-on-chip [18]. A time-predictable memory controller supports analyzable access to main memory [4]. The compiler for Patmos is an adaption of the LLVM compiler infrastructure [14]. Furthermore, AbsInt's WCET tool aiT [7] was adapted to support Patmos.

B. Patmos

The Patmos processor core is designed as a VLIW architecture and consists of five pipeline stages in the classical RISC style [20]. As the main goal of Patmos is to be highly time-predictable, all components are built so that they can be modeled efficiently for the WCET analyzer. The memory hierarchy consists of different caches suited for WCET analysis.

Some memory accesses to specific types of data such as instructions or stack frames allow for a more precise static cache analysis than in the general case. Directing such accesses through dedicated caches prevents interference from unpredictable memory accesses in the static analysis and thus

[1] see http://www.t-crest.org/

improves the WCET analysis. The split cache architecture of Patmos supports stack-allocated data [1] and heap-allocated data with two independent caches. For the instruction cache, the method cache, as presented in this paper, is used to achieve a time-predictable memory hierarchy.

The Patmos processor profits from the customized compiler support [14]. For example, the compiler splits oversized functions to fit them into the method cache. This is performed by a function splitter, which ensures a correct splitting of functions into smaller sub-functions.

C. Simulator

At the start of the development of Patmos, a cycle-accurate software simulator was developed within the T-CREST project [3]. 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.

D. Chisel

We use Chisel [2] 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.

E. Contributions

This paper presents the method cache design, a hardware implementation of the cache, and its integration into Patmos. Furthermore, the paper describes method cache variations, and compares them with respect to hardware resources and performance. A simulation model of the processor verifies the hardware results.

This paper is organized in six sections. Section II presents related work. Section III presents the basic idea of a method cache and variations of it. Section IV presents the design and implementation of the method cache for Patmos. Section V evaluates the proposed design and compares different method cache design. Section VI concludes the paper.

II. Related Work

A method cache with a first-in/first-out (FIFO) replacement strategy was first implemented and tested in the Java optimized processor (JOP) [16]. Within the JOP project, WCET analysis for the proposed method cache was studied in detail [19]. An IPET-based analysis and a variant using model checking are presented. The IPET analysis models the method cache FIFO replacement policy by bounding by the number of different blocks accessed during the execution of a given code region. If the method cache has the capacity to hold N blocks in the memory and a block is loaded to the cache, it stays there for at least N further cache misses and is therefore loaded at most once if at most N different cache blocks are accessed during an execution of a scope. The model-checking-based WCET analysis leads to tighter (between 2% and 7%) WCET bounds than the IPET-based WCET analysis [19]. However, the analysis runtime with model checking is prohibitively high.

Metzlaff et al. propose a dynamically managed instruction scratchpad (D-ISP) to increase the predictability of the processor architecture [9]. The instructions are loaded into the scratchpad memory at run-time, where the function start is used as a “reload point”. The D-ISP loads whole functions and is therefore similar to a method cache.

A hardware implementation of the D-ISP and an evaluation of its average-case performance is presented in [8]. The evaluation compares the D-ISP to a conventional direct-mapped cache and a statically managed instruction scratchpad, using three different benchmark suites. Results show that the D-ISP performs slightly worse than a conventional instruction cache but better than a static scratchpad.

Metzlaff and Ungerer present a comparison between different replacement strategies for the D-ISP with regard to the WCET [10]. FIFO-replacement, LRU-replacement, and stack replacement policies are compared. Timing analysis results show that the stack and LRU replacement perform better than FIFO, although the stack approach lacks performance when the scratchpad size increases.

An extension of the analysis of the D-ISP is presented by Metzlaff and Ungerer in [11]. The work compares different on-chip memory allocation strategies with regard to their WCET properties. An instruction cache (fully associative with LRU replacement) is compared to a statically allocated instruction scratchpad and a D-ISP. The evaluation shows a better WCET estimation for the D-ISP compared to the other solutions, especially with an increasing size of the on-chip memory. One major benefit is that the D-ISP avoids interference from unpredictable off-chip memory accesses. Therefore better memory access times can be assumed in the analysis. The paper also evaluates different off-chip memory latencies. An increasing memory latency favors the D-ISP compared to the other approaches.

Preußer et al. propose a stack-oriented variant of a method cache [13] for the SHAP processor [21]. Results show that the stack-oriented allocation policy provides average-case performance similar to a method cache using FIFO replacement.

III. The Method Cache and Variations

A method cache is organized to cache full functions [15]. The concept was originally developed for JOP [16], which executes Java bytecode. As Java uses the term “method”, the cache is called “method” cache, but is also applicable for functions and procedures.

A function may be loaded into the cache on a call and on a return when the caller has been evicted from the cache. On a call, the cache checks whether the called function is in the cache. If it is a miss, the function is loaded into the cache. On a return, the caller is checked and loaded on a miss. The advantage is that all other instructions are guaranteed hits and can be ignored by the cache analysis. Only the call tree needs to be considered to analyze cache hits and misses. Furthermore, cache entries never conflict due to their addresses in the method cache, eliminating the need for code placement optimizations.

In this paper we present two variants of the method cache: (1) the fixed-block method cache and (2) the variable-size method cache.

The basic organization of the fixed-block method cache is a local memory that is divided into blocks. A loaded function can span multiple blocks, but the function needs to be loaded into contiguous blocks. This organization allows two replacement policies: (1) first-in/first-out (FIFO), which works like a ring buffer, and (2) a stack-oriented replacement where the allocation of blocks follows the same regime as the allocation of stack frames. However, the stack-oriented replacement leads to conflicts for functions at the same depth in the call tree. This conflict results in continuous replacement of functions when called in a loop.

For a fixed-block method cache, a tag entry is associated with each block. The entry for the first block of the function contains the tag entry (address of the function in the memory). The tags for other blocks of the function are cleared on a function load. With FIFO replacement, this mechanism automatically removes the tag entry of a function when the first block of a function is overwritten by a newly loaded function.

A variation of the method cache is to decouple the tag memory organization from cache blocks. This allows to reduce the block size to single instruction words and to use every tag memory entry for a different function. We call this organization variable-size method cache. The D-ISP is organized in a similar way [8], [9]. With this variant, the tag memory also contains the position of the function in the cache and the size of the function. The number of tag entries limits the number of functions that can be in the cache at the same time.

With a variable-size method cache, the cache memory is better utilized, because functions are loaded back to back. Only a single area in the memory remains that might be unused. For the fixed-block method cache, each loaded function has one block that might contain unused space.

A function is always loaded into the cache as a whole. Therefore, functions must not exceed the cache size. The Patmos compiler ensures this property by splitting large functions into smaller functions.

IV. Method Cache Design and Implementation

Figure 1 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.

A. 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.

B. Hardware Implementation

We have implemented the method cache in Chisel [2], 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 sub-components 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.

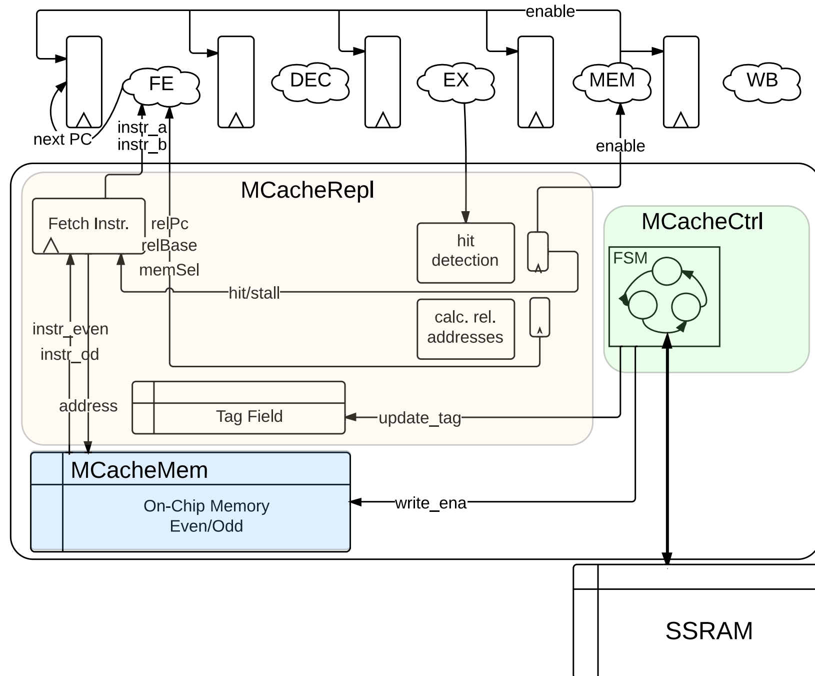


Fig. 1. The method cache and the relation to the processor pipeline and the external memory

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.

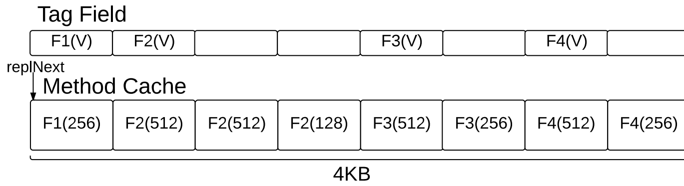
The `MCacheCtrl` component implements the transfer from the external memory to the on-chip cache memory. A finite state machine (FSM) handles the correct sequence of communication between the external memory controller and the method cache. As long as the replacement unit reports a hit, the controller stays idle. A miss causes the unit to load the requested function from the connected external memory. A subset of the OCP protocol [12] 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 bust 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.

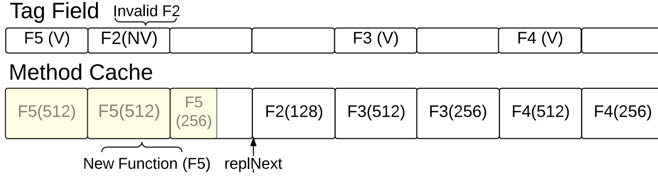
C. Method Cache Variations

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.

1) Fixed-Block Method Cache: A simple implementation of a replacement strategy for the method cache is to use FIFO

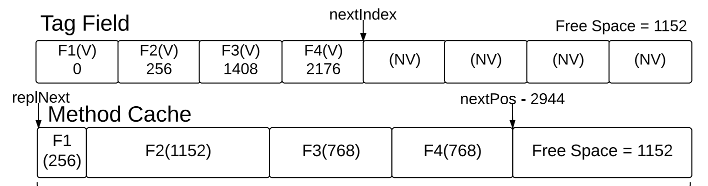


(a) Method cache filled with functions F1-F4

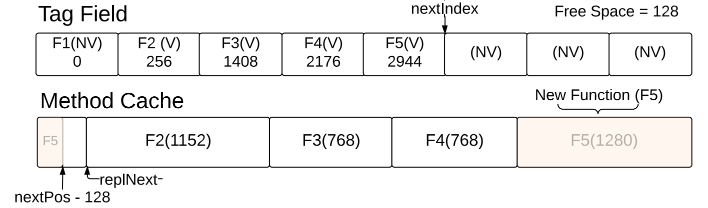


(b) F5 replaces F1 and F2

Fig. 2. Fixed-block method cache with FIFO replacement



(a) Method cache filled with functions F1-F4



(b) F5 fills free space and replaces F1

Fig. 3. Variable-size method cache with FIFO replacement

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 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:

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

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 2 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.

2) Variable-Size Method Cache: A basic problem of the replacement with fixed blocks is the waste of memory in blocks which 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 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 3 shows an example of a variable-size method cache. The cache is 4 KB large and its tag memory can hold up to

TABLE I
HARDWARE COSTS OF A 4 KB METHOD CACHE IN LOGIC CELLS

# functions	LC (V)	LC (F)
4	1098	886
8	1445	1054
16	2083	1369
32	3381	2025

eight references to functions. As in the example in Figure 2, four functions, F1 to F4, are already loaded into the cache (a). In contrast to the example in Figure 2, 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 2, where two functions in the cache have to be replaced.

D. Hardware Cost

Table I 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.

V. EVALUATION

We evaluate the proposed method cache with embedded benchmarks from the Mälardalen benchmark suite [5] and from the MiBench benchmark suite [6], 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.

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.

A. Evaluation Methodology

We have three evaluation platforms available: (1) the processor executing in real hardware (in an FPGA), (2) the emulator generated from the Chisel hardware description, and (3) the software simulator for Patmos. Each of those platforms has different properties in usage and confidence in the results.

The processor executing in an FPGA is the platform that delivers the exact results, as this is the current target device. However, getting performance data out of the hardware is cumbersome. To check results, we added counters for cache hits, cache misses, and execution cycles to the processor. These counters are connected to LEDs on the FPGA board. With this setup we compared the output of the hardware with the output of the emulator in 10 different configurations. The results in the hardware were equal with the results in the emulator, which confirms that there are no unexpected side-effects of the simplified boot procedure in the emulator. Therefore, we continue our evaluation with the emulator.

The emulator is generated from the same hardware description as the hardware and is therefore cycle accurate. The emulator is generated C++ code and easy to instrument for measurements. Therefore, we use it extensively for the evaluation. However, each cache configuration requires regeneration and recompilation of the emulator. Therefore, we also use the simulator for the evaluation.

The simulator of Patmos is a cycle-accurate C++ implementation of Patmos. This software simulator is highly configurable (e.g., cache sizes, external memory latencies) and delivers detailed simulation results (e.g., execution time and cache hits/misses, but also profiling information, cache utilization and method cache size requests). Therefore, it is convenient for detailed explorations of the method cache. However, the software simulator is only a model of the real platform. In order to prevent any deviation of the behavior of the simulator from the hardware implementation, we use the emulator and the simulator together to validate the results.

B. Function Inlining and Splitting

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

TABLE II
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 %

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.

C. Performance

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 14 functions (including floating point library functions) that are executed, whose sizes vary between 72 bytes and 2.3 KB, amounting to a total of 10.6 KB of code. The default compiler setup splits the program into 48 sub-functions of an average size below 256 bytes.

1) Variable-Size versus Fixed-Block Method Cache: Table II 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.

2) Compiler Settings: Table III and Table IV 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

TABLE III
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

TABLE IV
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

functions that are active 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 III shows the results with a compiler setting that splits code into sub-functions of up to 1 KB. Table IV 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 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 IV 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 most 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 IV, where for 32 functions, increasing

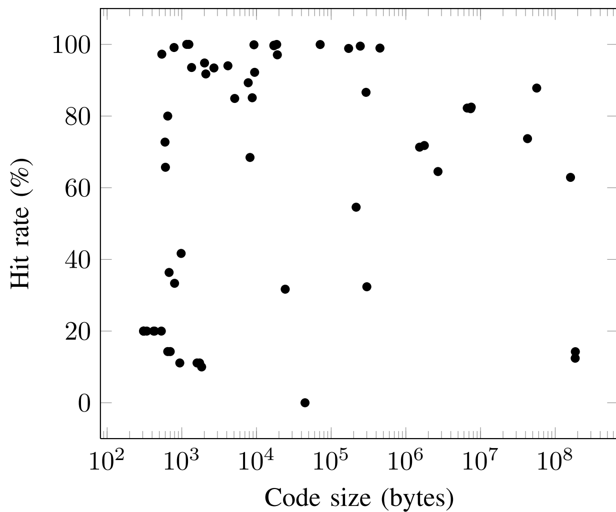


Fig. 4. Hit rate over code size for 59 different benchmarks for a 8 KB method cache with maximum 32 methods

the cache size beyond 8 KB has no effect, and for smaller tag memories doubling the 4 KB cache has no effect.

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 [5] and the MiBench benchmark [6] suites.

Figure 4 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 5 shows a bar graph with the hit rate for each evaluated benchmark.[2] For the benchmarks close to 100%, 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 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%.

VI. Conclusion

Real-time systems need time-predictable comport architectures. In this paper we presented a time-predictable instruction cache; the method cache. The method cache caches whole functions and therefore misses can only happen at call and return instructions. This leads to a simplification of the worst-case execution time analysis.

We implemented the method cache in the time-predictable processor Patmos and evaluated the performance of different configurations. The comparison of the fixed-block and variable-size method cache shows an improvement of the hit/miss ratio for the latter one. Different compiler settings demonstrate the dependency of the cache performance on the function-splitting by the compiler. Results show an improved performance when splitting into smaller sub-functions.

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

Acknowledgment

We thank Florian Brander for the development of the Patmos software simulator, a tool that has helped design and development activity within the T-CREST project.

This work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). This work is part of the project "Hard Real-Time Embedded Multiprocessor Platform - RTEMP" and received partial funding from the Danish Research Council for Technology and Production Sciences under contract 12-127600.

Source Access

We provide the code of the method cache, included in Patmos, and the compiler for Patmos in open source at:
https://github.com/t-crest

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] 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.

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

[4] 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.

[5] 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.

[6] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

[7] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. Technical report, AbsInt Angewandte Informatik GmbH. [Onlinei (http://www.absint.de/aiT_WCET.pdf), last accessed November 2013].

[8] Stefan Metzlaff, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In Mladen Berekovic, William Fornaciari, Uwe Brinkschulte, and Cristina Silvano, editors, *Architecture of Computing Systems - ARCS 2011*, volume 6566 of *Lecture Notes in Computer Science*, pages 122–134. Springer Berlin / Heidelberg, 2011.

[9] Stefan Metzlaff, Sascha Uhrig, Jörg Mische, and Theo Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *Proceedings of the 9th workshop on Memory performance (MEDEA 2008)*, pages 38–45, New York, NY, USA, 2008. ACM.

[10] Stefan Metzlaff and Theo Ungerer. Impact of instruction cache and different instruction scratchpads on the WCET estimate. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1442 –1449, june 2012.

Fig. 5. Hit rate for 59 different benchmarks for a 8KB method cache with maximum 32 methods

[11] Stefan Metzlaff and Theo Ungerer. A comparison of instruction memories from the WCET perspective. *Journal of Systems Architecture*, pages –, 2013.

[12] OCP International Partnership. Open Core Protocol specification, release 3.0, 2009.

[13] Thomas B. Preusser, Martin Zabel, and Rainer G. Spallek. Bump-pointer method caching for embedded Java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.

[14] 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.

[15] 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.

[16] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[17] Martin Schoeberl. Is time predictability quantifiable? In *International Conference on Embedded Computer Systems (SAMOS 2012)*, Samos, Greece, July 2012. IEEE.

[18] 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.

[19] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

[20] 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.

[21] Martin Zabel and Rainer G. Spallek. SHAP — scalable multi-core Java bytecode processor. Technical report, Fakultät Informatik, Technische Universität Dresden, 2009. ftp://ftp.inf.tu-dresden.de/pub/berichte/tud09-13.pdf.