Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach

Martin Schoeberl[1], Pascal Schleuniger[1], Wolfgang Puffitsch[2], Florian Brandner[3], Christian W. Probst[1], Sven Karlsson[1], and Tommy Thorn[4]

1 **Department of Informatics and Mathematical Modeling**
Technical University of Denmark
masca@imm.dtu.dk, pass@imm.dtu.dk, probst@imm.dtu.dk, ska@imm.dtu.dk
2 **Institute of Computer Engineering**
Vienna University of Technology, Austria
wpuffits@mail.tuwien.ac.at
3 **COMPSYS, LIP, ENS de Lyon**
UMR 5668 CNRS – ENS de Lyon – UCB Lyon – Inria
florian.brandner@ens-lyon.fr
4 **Unaffiliated Research**
California, USA
tommy@thorn.ws

Abstract

Current processors are optimized for average case performance, often leading to a high worst-case execution time (WCET). Many architectural features that increase the average case performance are hard to be modeled for the WCET analysis. In this paper we present Patmos, a processor optimized for low WCET bounds rather than high average case performance. Patmos is a dual-issue, statically scheduled RISC processor. The instruction cache is organized as a method cache and the data cache is organized as a split cache in order to simplify the cache WCET analysis. To fill the dual-issue pipeline with enough useful instructions, Patmos relies on a customized compiler. The compiler also plays a central role in optimizing the application for the WCET instead of average case performance.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems – Real-time and embedded systems
C1.1 Processor Architectures – Single Data Stream Architectures – RISC/CISC, VLIW architectures

Keywords and phrases Time-predictable architecture, WCET analysis, WCET-aware compilation

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

Real-time systems need a time-predictable execution platform so that the worst-case execution time (WCET) can be estimated statically. It has been argued that we have to rethink computer architecture for real-time systems instead of trying to catch up with new processors in the WCET analysis tools [21, 3, 23].

However, time-predictable architectures alone are not enough. If we would only be interested in time predictability, we could use microprocessors from the late 1970s to the mid-1980s, where the execution time was accurately described in the data sheets. With those processors it would be possible to *generate* exact timing in software, e.g., one of the authors has programmed a wall clock on the Zilog Z80 in assembler by counting instruction clock cycles and inserting delay loops and nops at the correct locations.

First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 1–10

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Processors for future embedded systems need to be time-predictable *and* provide a reasonable worst-case performance. Therefore, we present a Very Long Instruction Word (VLIW) pipeline with specially designed caches to provide good single thread performance. We intend to build a chip-multiprocessor using this VLIW pipeline to investigate its benefits for multi-threaded applications.

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 tool and helps to reduce the overestimation caused by imprecise information. Memory hierarchies having multiple levels of caches typically pose a major challenge for the WCET analysis. We attack this issue by introducing caches that are specifically designed to support WCET analysis. For instructions we adopt the method cache, as proposed in [18], which operates on whole functions/methods and thus simplifies the modeling for WCET analysis. Furthermore, we propose a split cache architecture for data [20], offering dedicated caches for the stack area, for constants and static data, as well as for heap allocated objects. A compiler-managed scratchpad memory provides additional flexibility. Specializing the cache structure to the usage patterns of its data allows predictable and effective caching of that data, while at the same time facilitating WCET analysis.

Aside from the hardware implementation of Patmos, we also present a sketch of the software tools envisioned 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 instruction set level. Hard to predict features 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 plan to provide a *WCET-aware* software development environment tightly integrating traditional WCET tools and compilers. The heart of this environment is a *WCET-aware* compiler that is able to preserve annotations for WCET analysis, actively optimize the WCET, and exploit the specialized architectural features of Patmos.

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. [9] 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. This includes, but is not limited to, time-predictable caching mechanisms, chip-multiprocessing (CMP), as well as novel pipeline organizations. Patmos is open-source under a BSD-like license.

The presented processor is named after the Greek island Patmos, where the first sketches of the architecture have been drawn; not in sand, but in a (paper) notebook. If you use the open-source design of Patmos for further research, we would suggest that you visit and enjoy the island Patmos. Consider writing a postcard from there to the authors of this paper.

The paper is organized as follows: In the following section related work on time-predictable processor architectures and WCET driven compilation is presented. The architecture of Patmos is described in Section 3, followed by the proposal of the software development tools in Section 4. The experience with initial prototypes of the processor and a compiler backend is reported in Section 5 and the paper is concluded in Section 6.

2 Related Work

Edwards and Lee argue: “It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [3]. A first simulation of their PRET architecture is presented in

[12]. PRET implements a RISC pipeline and performs chip-level multi-threading for six threads to eliminate data forwarding and branch prediction. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a time-division multiple access (TDMA) scheme, called memory wheel. The ISA is extended with a *deadline* instruction that stalls the current thread until the deadline is reached. This instruction is used to perform time-based, instead of lock-based, synchronization for accesses to shared data. Furthermore, it has been suggested that the multi-threaded pipeline explores pipelined access to DRAM memories [2]. Each thread is assigned its own memory bank.

Thiele and Wilhelm argue that a new research discipline is needed for time-predictable embedded systems [23]. Berg et al. identify the following design principles for a time-predictable processor: "... recoverability from information loss in the analysis, minimal variation of the instruction timing, non-interference between processor components, deterministic processor behavior, and comprehensive documentation" [1]. The authors propose a processor architecture that meets these design principles. The processor is a classic five-stage RISC pipeline with minimal changes to the instruction set. Suggestions for future architectures of memory hierarchies are given in [26].

Time-predictable architectural features have been explored in the context of the Java processor JOP [19]. The pipeline and the microcode, which implements the instruction set of the Java Virtual Machine, have been designed to avoid timing dependencies between bytecode instructions. JOP uses split load instructions to partially hide memory latencies. Caches are designed to be time-predictable and analyzable [18, 20, 22, 9]. With Patmos we will leverage on our experience with JOP and implement a similar, but more general, cache structure.

Heckmann et al. provide examples of problematic processor features in [8]. The most problematic features found are the replacement strategies for set-associative caches. In conclusion Heckmann et al. suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches; (2) locally deterministic update strategies for caches; (3) static branch prediction; and (4) limited out-of-order execution. The authors argue for restriction of processor features. In contrast, we also provide additional features for a time-predictable processor.

Whitham argues that the execution time of a basic block has to be independent of the execution history [24]. To reduce the WCET, Whitham proposes to implement the time critical functions in microcode on a reconfigurable function unit (RFU). With several RFUs, it is possible to explicitly exploit instruction level parallelism (ILP) of the original RISC code – similar to a VLIW architecture.

Superscalar out-of-order processors can achieve higher performance than in-order designs, but are difficult to handle in WCET analysis. Whitham and Audsley present modifications to out-of-order processors to achieve time-predictable operation [25]. Virtual traces allow static WCET analysis, which is performed before execution. Those virtual traces are formed within the program and constrain the out-of order scheduler built into the CPU to execute deterministically.

An early proposal [17] of a WCET-predictable super-scalar processor includes a mechanism to avoid long timing effects. The idea is to restrict the fetch stage to disallow instructions from two different basic blocks being fetched in the same cycle. For the detection of basic blocks in the hardware, additional compiler inserted branches or special instructions are suggested.

Multi-Core Execution of Hard Real-Time Applications Supporting Analyzability (MERASA) is a European Union project that aims for multicore processor designs in hard real-time embedded systems. An in-order superscalar processor is adapted for chip multi-threading (CarCore) [14]. The resulting CarCore is a two-way, five-stage pipeline with separated address and data paths. This architecture allows issuing an address and an integer instruction within one cycle, even if they are data-dependent. CarCore supports a single hard real-time thread to be executed with several non-real-time threads running concurrently in the background.

In contrast to the PRET and CarCore designs we use a VLIW approach instead of chip-level

multi-threading to utilize the hardware resources. To benefit from thread-level applications we will replicate the simple pipeline to build a CMP system. For time-predictable multi-threading almost all resources (e.g., thread local caches) need to be duplicated. Therefore, we believe that a CMP system is more efficient than chip multi-threading.

Compilers trying to take the WCET into account have been subject of intense research. A major challenge is to keep annotations, intended to aid the WCET analysis, up-to-date throughout the optimization and transformation phases of the compiler. So far, techniques are known to preserve annotations for a limited set of compiler optimizations [4, 10] only. A more directe approach to WCET-aware optimization is offered by the WCC compiler of Falk et al. [13, 5, 6]. Here, optimizations are evaluated using a WCET analysis tool and only applied when shown to be beneficial. A similar approach is taken by Zhao et al. [27], where a WCET-analysis tool provides information on the critical paths which are subsequently optimized. These efforts only represent a first step towards developing WCET-aware compilation techniques by discarding counter productive optimization results. A disciplined approach for the design of true WCET-aware optimizations is, however, not known and still considered an open problem.

3 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 VLIW architecture was choosen. For multi-threaded code we plan to build a chip-multiprocessor system with statically scheduled access to shared main memory [15].

Patmos is a statically scheduled, dual-issue RISC microprocessor. The processor does not stall, except for explicit instructions that wait for data from the memory controller. 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 and efficient code. Programming Patmos is consequently more demanding than for usual processors. However, knowing all 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 tasks 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 [9]. Therefore, Patmos contains several data caches, one for each memory area. Furthermore, we will explore the benefits of compiler managed scratchpad memory.

The primary implementation technology is in a field-programmable gate array (FPGA). Therefore, the design is optimized within the technology constraints of an FPGA. Nevertheless, features such as preinitialized on-chip memories are avoided to keep the design implementable in ASIC technologies.

3.1 Instruction Set

The instruction set of Patmos follows the conventions of usual RISC machines such as MIPS. All instructions are fully predicated and take at most three register operands. Except for branch and accesses to main memory using loads or stores, all instructions can be executed by both pipelines.

The first instruction of an instruction bundle contains 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 constant fields and branch offsets. Constants are supported in different ways. A few ALU instruction can be performed with a sign-extended 12-bit constant operand. Two instructions are available to load 16 bits into the lower (with

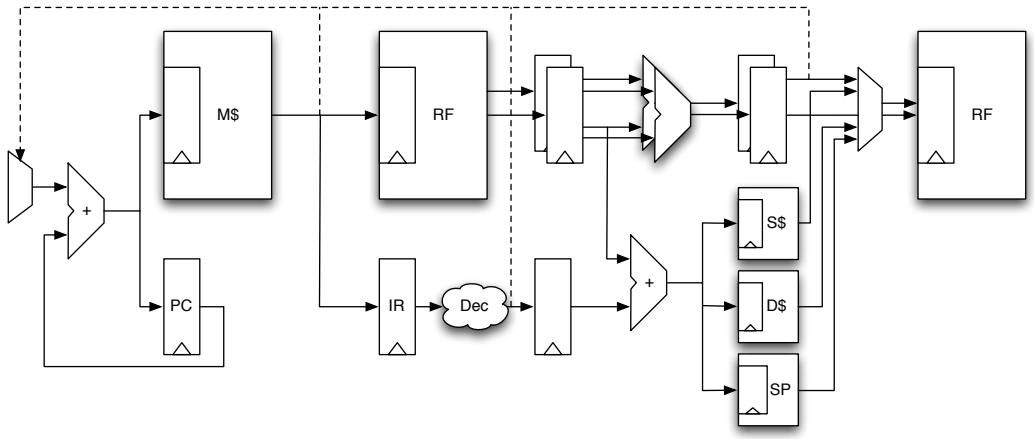

Figure 1 Pipeline of Patmos with fetch, decode, execute, and memory/write back stages.

sign extension) or upper half of a register. Furthermore, a 32-bit constant can be loaded into a register by using the second instruction slot for the constant. Branches (conditional and unconditional) are relative with a 22-bit offset. Function calls to a 32-bit address are supported by a register indirect branch and link instruction.

To reduce the number of conditional branches and to support the single-path programing paradigm [16], Patmos supports fully predicated instructions. Predicates are set with compare instructions, which itself can be predicated. A complete set of compare instructions (two registers and register against 0) is supported. The optimum number of concurrently live predicates is still not settled, but will be at least 8.

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.

3.2 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 (on-chip) memory access and the register write back is merged into a single stage. The data cache is split into different cache areas. The distinction between the different caches is performed with typed load and store instructions.

Figure 1 shows an overview of Patmos' pipeline. To simplify the diagram, forwarding and external memory access data paths are omitted and not all typed caches are shown. 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 can not 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.

For a dual-issue RISC, the RF needs four read ports and two write ports. Current FPGAs offer on-chip memories with one read and one write port. Additional read ports can be implemented by replicating the RF on several on-chip memories. However, to implement the dual write ports, the RF needs to be double clocked. To save resources, double clocking is also used for the read ports. The

resulting RF needs *only* two block RAMs. As read during write at the same address in the on-chip memories of current FPGAs either delivers the old value on the read or an undefined value the RF contains an internal forwarding path.

At the execution stage up to two operations are executed and the address for a memory access is calculated. Predicates are set on a compare instruction. The last stage writes back the results from the execution stage or loads data from one of the data cache areas.

The PC manipulation depends on three pipeline stages, as sketched with the dashed line in Figure 1. At the fetch stage the single bit that determines the instruction length is fed to the PC multiplexer. Unconditional branches are detected at the decode stage and the branch offset is fed to the multiplexer from IR. The predicate for a conditional branch is available as a result from the execution stage and the PC multiplexer also depends on the write back stage.

3.3 Memory and Caches

Access to main memory is done via a split load, where one instruction starts the memory read and another instruction explicitly waits for the result. Although this increases the number of instructions to be executed, instruction scheduling can use the split accesses to hide memory access latencies deterministically. For instruction caching a method cache is used where full functions/methods are loaded at call or return [18]. This cache organization simplifies the pipeline and the WCET analysis as instruction cache misses can only happen at call or return instructions. For the data cache a split cache is used [20]. Data allocated on the stack is served by a direct mapped stack cache, heap allocated data in a highly associative data cache, and constants and static data in a set associative cache. Only the cache for heap allocated data and static data needs a cache coherence protocol for a CMP configuration of Patmos. Furthermore, a scratchpad memory can also be used to store frequently accessed data. To distinguish between the different caches, Patmos implements typed load and store instructions. The type information is assigned by the compiler (e.g., the compiler already organizes the stack allocated data). To simplify Figure 1, only the stack and data cache are shown as an example of the split cache.

4 Software Development with Patmos

The architecture design of Patmos adopts ideas from the RISC and VLIW design-philosophies. In particular, the idea that architecture design is *interdependent* on the software development environment. The first RISC machines made some architectural constraints visible on the instruction set level in order to push complexity from the hardware design to the software tools or programmer. The VLIW philosophy took this idea even further and assigned the compiler a central role in exploiting the available hardware resources in the best possible way [7].

We make the case that this architecture philosophy is particularly suited to address the problems encountered in today's real-time system design. Time-predictable architectures following this approach, such as Patmos, not only unveil optimization potential to the compiler, but more importantly provide the opportunity for developing more accurate program analyses, e.g., in order to derive tighter bounds for the WCET. The compiler and the program analysis tools are thus first class citizens of the real-time system engineer's toolbox and need to be accounted for in the architecture design. As a side-effect the use of high-level programming languages is facilitated or even favored, since the necessary software tools are readily provided.

4.1 WCET-aware Compilation

The Patmos approach relies on a strong compiler in order to optimally exploit the available hardware resources. Traditionally, compilers seek to optimize the *average execution time* by focusing the effort on frequently executed *hot paths*. For other, rarely executed, code paths a performance degradation is usually acceptable. This view of a compiler and its optimizations is *not* valid in our context. But, what is the compiler supposed to optimize then? And how could such a compiler look like?

The WCET is an important metric in order to determine whether a real-time program can be scheduled and meets its deadlines. The actual WCET is in fact rarely known but instead approximated by a WCET bound, which is usually provided by a program analysis tool independent from the compiler. The WCET or its bound are suitable candidates as a primary optimization goal for our compiler. Their optimization, however, poses some difficult problems that need to be addressed in the future, opening up a new field for compiler researches and architecture designers.

Foremost, the compiler has to be aware of the WCET. We will consequently integration the WCET analysis tools tightly with the compiler. In practice, we expect synergetic effects from this integration, as both tools usually share a great deal of infrastructure. Most importantly, the WCET analysis is likely to profit from additional information that is available from the compiler throughout the translation process from a high-level input program to its machine form. The preservation of relevant information required by the WCET analysis, in particular annotations provided by the programmer, is a major challenge that has only been solved for selected code transformations [10].

In addition, a new approach to compilation is needed that focuses on optimizing the *critical paths* of a program instead of its hot paths [6, 27]. However, the critical paths may change during the optimization process, either because the previous critical path has been sped-up or because the *optimization* adversely affected another path slowing it down. This gives rise to *phase-ordering* problems throughout the optimization process. The problem here is to decide which code regions are to be optimized and in which order. In addition, optimizations may adversely effect each other, such that the relative ordering of optimizations needs to be accounted for in a WCET-aware compiler. Defining a sound optimization strategy for a WCET-aware compiler is still considered to be an open problem. A key insight is that a time-predictable architecture is mandatory for defining such an optimization strategy. It becomes otherwise impossible to asses the impact of a given transformation on the WCET, resulting in the application of undesirable *optimizations*, inefficient code, and consequently conservative WCET-bounds.

4.2 Exploiting Patmos' Features

Some design decisions for Patmos are based on a pragmatic assumption that the engineer best knows the system under development. It is thus important to enable the programmer to fine tune the system. Care has been taken that those features are *accessible* from high-level programming languages. The typed memory loads and stores are a good example of such a feature, which allows the programmer to explicitly assign variables and data structures to specific storage elements. The typed memory operations are a natural match to named address spaces in Embedded C, an extension of the traditional C language. The computation of tight WCET bounds is simplified, since the target memory is apparent from the operation itself. The tedious tracking of possible pointer ranges is thus avoided.

The stack cache provides a time-predictable and analyzable way to reduce the penalty for accessing objects residing on the stack frame of the current function. For most functions it is trivial for the compiler to immediately exploit the stack cache. Special care has to be taken that function-local variables accessible through pointers are not placed in the cache, because the cache's memory is not accessible using regular memory operations. Those variables need to be kept in a *shadow stack* residing in general purpose memory. Note that other variables of the same function are nevertheless

assigned to the stack cache.

Exploiting the method cache is more involved and requires a global analysis of the complete real-time program, including all external modules and libraries linked to it. Using a regular call graph we can determine function calls potentially leading to conflicts in the cache and adopt the placement of the involved functions accordingly. Similar techniques have successfully been applied in the context of scratchpad memories and overlay memories [5]. The design of Patmos' method cache, however, combines the predictability of a static code layout in a scratchpad memory with the flexibility of a cache.

The predicated instructions supported by Patmos allow the elimination of branches. This idea was first applied for wide-issue VLIW machines in order to keep the parallel execution units busy and avoid the expensive branch penalty. The single-path programing paradigm [16] adopts the very same idea to compute tighter WCET bounds. While it is true that for a given single-path program the WCET bound is generally closer to the actual WCET, the absolute WCET and its computable bound is *not* guaranteed to be better than for regular programs. The problem arises from the blind elimination of branches independent from their relevance to the final WCET. We thus propose WCET-aware if-conversion and global scheduling in order to eliminate branches and exploit the parallel execution units of Patmos to actively reduce the absolute WCET.

5 Evaluation

To evaluate Patmos we are working in parallel on the following pieces: a SystemC simulation model, a VHDL-based FPGA implementation, a port of the GNU Binutils and the LLVM compiler [11].

A VHDL hardware prototype was implemented to get an idea on the speed of the system and to evaluate the feasibility of a time division multiplexed register file. For that reason two parallel RISC pipelines, with common instruction fetch stage and shared register file and data cache were implemented. The single pipelines are based on a load/store architecture that uses write back.

Modern FPGAs contain extensive memory resources in terms of block RAMs. Those SRAM-blocks can often be clocked with frequencies higher than 500 MHz. The register file in a VLIW architecture requires a multi-port RAM that provides simultaneous access to four read and two write ports. Previous soft core implementations have shown that the resulting system clock frequency is far below the clocking capabilities of block RAMs. For that reason it seems natural to access memory time division multiplexed. This allows making use of the fast clocking capabilities of the block RAMs and is less hardware resource demanding than a classical multi-port memory implementation.

On the downside, using multiple clocks in a pipeline implies timing problems that might require a slowdown of the system clock frequency. Simulation on the hardware model showed that the performance of the system greatly depends on the quality of the clocks. When the two clocks were derived from an accurate PLL unit, a maximum pipeline clock frequency of more than 200 MHz on a Xilinx Virtex 5 (speed grade 2) can be reached. The ALU unit remained the critical path.

It can be concluded that the use of double-clocked block RAM for the register file in VLIW architectures is an appropriate solution to exploit the available resources of modern FPGAs. The promising results motivate to pursue the chosen track and to implement the remaining functionality of the Patmos soft core.

As compiler we adapted LLVM [11] to support the instruction set of Patmos. For most parts of the compiler backend, the proposed architecture can be treated as plain RISC architecture. Due to the open-source nature of LLVM, it is possible to reuse code from existing backends with similar characteristics. A first rough port for Patmos has been implemented within a few days, by picking appropriate code from the other backends. A feature that differs from other instruction sets is the splitting of memory accesses. However, LLVM provides means to customize the instruction selection

in the backend appropriately, without changing the core code.

Where a VLIW *does* differ significantly from a RISC architecture is instruction scheduling. Two instructions can be scheduled per cycle, and appropriate markers to separate instruction bundles have to be inserted. Due to the simplicity of the proposed architecture, we believe that one of the existing instruction schedulers in LLVM can be reused for our architecture with modest customization.

6 Conclusion

In this paper we presented the time-predictable processor Patmos. We believe that future embedded real-time systems need processors designed to minimize the WCET and implement architectural features that are WCET analyzable. To provide good single thread performance Patmos implements a statically scheduled, dual-issue pipeline. With a first prototype we have evaluated the feasibility to implement a dual-issue processor in an FPGA without hurting the maximum clock frequency. Patmos will serve as platform for future research on co-development of time-predictable architecture features and their WCET analysis.

References

1 Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

2 Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*, Lake Tahoe, CA, October 2009. IEEE.

3 Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

4 Jakob Engblom. Worst-case execution time analysis for optimized code. In *In Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 146–153, 1997.

5 Heiko Falk and Jan C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *DAC '09: Proceedings of the Conference on Design Automation*, pages 732–737, 2009.

6 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, pages 1–50, 2010.

7 Joseph A. Fisher, Paolo Faraboschi, and Young Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

8 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

9 Benedikt Huber, Wolfgang Puffitsch, and Martin Schoeberl. WCET driven design space exploration of an object caches. 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.

10 Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1–2):72–105, June 2010.

11 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 75–88. IEEE Computer Society, 2004.

12 Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

13 Paul Lokuciejewski, Heiko Falk, and Peter Marwedel. WCET-driven cache-based procedure positioning optimizations. In *The 20th Euromicro Conference on Real-Time Systems (ECRTS 2008)*, pages 321–330. IEEE Computer Society, 2008.

14 Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. How to enhance a superscalar processor to provide hard real-time capable in-order smt. In *23rd International Conference on Architecture of Computing Systems (ARCS 2010)*, pages 2–14, University of Augsburg, Germany, February 2010. Springer.

15 Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

16 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.

17 Christine Rochange and Pascal Sainrat. Towards designing WCET-predictable processors. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003*, pages 87–90, 2003.

18 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.

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

20 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.

21 Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009.

22 Martin Schoeberl, Wolfgang Puffitsch, and Benedikt Huber. Towards time-predictable data caches for chip-multiprocessors. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*, number LNCS 5860, pages 180–191. Springer, November 2009.

23 Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

24 Jack Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, 2008.

25 Jack Whitham and Neil Audsley. Time-predictable out-of-order execution for hard real-time systems. *IEEE Transactions on Computers*, 59(9):1210–1223, 2010.

26 Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

27 Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by applying worst-case path optimizations. *Real-Time Systems*, 34:129–152, 2006.