Bringing Theory to Practice:

Predictability and Performance in Embedded Systems

PPES'11, March 18, 2011, Grenoble, France

Edited by
Philipp Lucas
Lothar Thiele
Benoît Triquet
Theo Ungerer
Reinhard Wilhelm

Editors

Philipp Lucas, Universität des Saarlandes, Germany phlucas@cs.uni-saarland.de
Lothar Thiele, ETH Zürich, Switzerland thiele@ethz.ch
Benoît Triquet, Airbus, France benoit.triquet@airbus.com
Theo Ungerer, Augsburg University, Germany ungerer@informatik.uni-augsburg.de
Reinhard Wilhelm, Universität des Saarlandes, Germany wilhelm@cs.uni-saarland.de

ACM Classification 1998
C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

ISBN 978-3-939897-28-6

Published online and open access by
Schloss Dagstuhl – Leibniz–Zentrum für Informatik gGmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
March, 2011.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.PPES.2011.i

ISBN 978-3-939897-28-6 **ISSN 2190-6807** **www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

We are happy to present the proceedings of the 2011 Workshop on Predictability and Performance in Embedded Systems, held in March 2011 in Grenoble, France, as a satellite event of the Conference on Design, Automation & Test in Europe (DATE).

The PPES workshop is concerned with critical hard real-time systems that have to satisfy both efficiency and predictability requirements. For example, an electronic controller for a safety-critical system in an automobile needs to react not only correctly to external inputs such as rapid deceleration or loss of grip, but also provably within a given time-span. This topic of *reconciling predictability and performance* has received much interest in recent years, in particular considering its growing relevance and complexity with the advent of multi-core systems with shared resources.

The advancements in these fields, however, have been discussed mostly in the standard venues (general conferences, workshops, journals). The aim of this workshop is twofold:

- to present the results achieved and tools developed by various researchers, in particular to industrial end users;
- and to present the industrial viewpoint on needs and challenges which need to be tackled for applicability.

To this end, the workshop comprises an invited presentation by Ottmar Bender of Cassidian Electronics on *Predictability and Performance Requirements in Avionics Systems*, a panel discussion on *Predictability and Performance in Industrial Practice*, and a number of paper presentations. In this first instance of the workshop, we received 14 submissions. After a careful review, 7 submissions covering various aspects of predictability and performance have been selected to appear in these proceedings. We would like to thank all authors for submitting their work to this first instance of the workshop despite the tight deadlines.

PPES was supported by

- *ArtistDesign*, the European Network of Excellence on Embedded Systems Design
- the *PREDATOR* project (Design for Predictability and Efficiency)
- the*MERASA* project (Multi-Core Execution of Hard Real-Time Applications Supporting Analysability)

The workshop is organised by: Philipp Lucas (Universität des Saarlandes), Lothar Thiele (ETH Zürich), Benoît Triquet (Airbus), Theo Ungerer (Augsburg University) and Reinhard Wilhelm (Universität des Saarlandes; chair). We were supported in the Program Committee by Pascal Sainrat (University of Toulouse), Sami Yehia (Thales), Wang Yi (Uppsala University) and Rafael Zalman (Infineon). Additional reviews were provided by David Black-Schaffer, Unmesh Dutta Bordoloi, Christian Bradatsch, Giorgio Buttazzo, Mamoun Filali, Mike Gerdes, Claire Maiza, Jörg Mische, Eric Noulard, Christine Rochange and Sascha Uhrig. Our thanks also go to Nicola Nicolici, the workshop chair of DATE, and to Bashir M. Al-Hashimi, general chair of DATE, for making this event possible.

Philipp Lucas, Reinhard Wilhelm
Saarbrücken, March 2011

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Software Structure and WCET Predictability*

Gernot Gebhard[1], Christoph Cullmann[1], and Reinhold Heckmann[1]

1 **AbsInt Angewandte Informatik GmbH**
Science Park 1, D-66123 Saarbrücken, Germany
info@absint.com

Abstract

Being able to compute worst-case execution time bounds for tasks of an embedded software system with hard real-time constraints is crucial to ensure the correct (timing) behavior of the overall system. Any means to increase the (static) time predictability of the embedded software are of high interest – especially due to the ever-growing complexity of such software systems. In this paper we study existing coding proposals and guidelines, such as MISRA-C, and investigate whether they simplify static timing analysis. Furthermore, we investigate how additional knowledge, such as design-level information, can further aid in this process.

1998 ACM Subject Classification B.2.2 [*Performance Analysis and Design Aids*]: Worst-case analysis

Keywords and phrases WCET Predictability, Embedded Software Structure, Coding Guidelines

Digital Object Identifier 10.4230/OASIcs.PPES.2011.1

1 Introduction

Embedded hard real-time systems need reliable guarantees for the satisfaction of their timing constraints. Experience with the use of static timing analysis methods and the tools based on them in the automotive and the avionics industries is positive. However, both, the precision of the results and the efficiency of the analysis methods are highly dependent on the predictability of the execution platform [3] and of the software run on this platform.

In this paper, we concentrate on the effect of the software on the time predictability of the embedded system. More precisely, we study existing software development guidelines that are currently in production use and identify coding rules that might ease a static timing analysis of the developed software. Such coding guidelines are intended to lead the developer to producing – among others – reliable, maintainable, testable/analyzable, and reusable software. Code complexity is also a key aspect due to maintainability and testability issues. However, the coding rules are not explicitly intended to improve the software predictability with respect to static timing analysis.

Based on our experience of analyzing automotive and avionics software, we provide additional means to increase software time predictability. Certain information about the program behavior cannot be determined statically just from the binary itself (or from the source code, if available). Hence, additional (design-level) knowledge about the system behavior would allow for a more precise (static) timing analysis. For instance, different operating modes of a flight control unit, such as *plane is on ground* and *plane is in air*, might lead to mutual exclusive execution paths in the software system. By using this knowledge, a

* The research reported herein has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement No 216008 (PREDATOR).

static timing analyzer is able to produce much tighter worst-case execution time bounds for each mode of operation separately.

Section 2 discusses related work. Section 3 briefly introduces static timing analysis and discusses challenges static timing analysis has to face. Section 4 investigates existing coding guidelines for their prospects to aid software predictability and discusses further means to increase the predictability of embedded software systems. Finally, Section 5 concludes this paper.

2 Related Work

The impact of the source code structure on time predictability has been subject to several research papers and projects respectively.

For instance, Puschner and Kirner propose a *WCET-oriented programming* approach [11] that aims at producing few or no input-data dependent code. Basically, the idea is to transform the software into a single-path program. To realize input-data dependent behavior of the code – this cannot be avoided for any piece of complex software – predicated operations shall be used[1]. A major drawback of the proposed code transformation is that in every possible execution context of a function or loop, the processor would have to always fetch the corresponding instructions, even if they would not be executed. Hence, the single-path paradigm actually impairs the worst-case behavior.

Thiele and Wilhelm investigate threats to time predictability and propose design principles that support time predictability [14]. Among others the authors discuss the impact of software design on system predictability. For example, the use of dynamic data structures should be avoided, as these are hard to analyze statically.

Wenzel et al. [15] discuss the possible impact of existing software development guidelines (DO-178B, MISRA-C, and ARINC 653) on the WCET analyzability of the software. Furthermore, the authors provide challenging code patterns, some of which, however, do not appear to cause problems for binary-level, static WCET analysis. For instance, calls to library functions do not necessarily impair the software's time predictability. The implementation and thus the binary code of the called function determines the time predictability, and not the fact of the function being part of a library. Nonetheless, the binary code of the library functions are required to be available to ensure a precise static worst-case execution time analysis if complex hardware architectures are being used. For ARINC 653 implementations that are truly modular this might not always be the case.

The purpose of the project COLA (Cache Optimizations for LEON Analyses)[2] was to investigate how software can achieve maximum performance, whilst remaining analyzable, testable, and predictable. COLA is a follow-on project to the studies PEAL and PEAL2 (Prototype Execution-time Analyzer for LEON), which identified code layout and program execution patterns that result in cache risks, so called *cache killers*, and quantified their impact. Among others, the COLA project produced cache-aware coding rules that are specifically tailored to increase the time predictability of the LEON2 instruction cache.

The project MERASA aimed at the development of a predictable and (statically) analyzable multi-core processor for hard real-time embedded systems. Bonenfant et al. [1] propose coding guidelines to improve the analyzability of software executed on the MERASA platform.

---

[1] Yet many embedded hardware architectures, as e.g. PowerPC, do not support predicated operations.

[2] Funded by the European Space Agency (ESA) under the basic Technology Research Programme (TRP), ESA/ESTEC Contract AO/1-5877/08/NL/JK.

Both static analysis and measurement-based approaches are considered. In principle, these coding guidelines correspond to the MISRA-C guidelines discussed in Section 4.2.

3 Static Timing Analysis

Exact worst-case execution times (WCETs) are impossible or very hard to determine, even for the restricted class of real-time programs with their usual coding rules. Therefore, the available WCET analyzers only produce WCET guarantees, which are safe and precise upper bounds on the execution times of tasks. The combined requirements for timing analysis methods are:

- *soundness* – ensuring the reliability of the guarantees,
- *efficiency* – making them feasible in industrial practice, and
- *precision* – increasing the chance to prove the satisfaction of the timing constraints.

Any software system when executed on a modern high-performance processor shows a certain variation in execution time depending on the input data, the initial hardware state, and the interference with the environment. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, bounds for the execution times of basic blocks are determined, from which bounds for the whole system's execution time are derived.

Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are – in part – responsible for the gap between WCET guarantees and observed upper bounds and between BCET guarantees and observed lower bounds. How much is lost depends on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software.

Despite the potential loss of precision caused by abstraction, static timing analysis methods are well established in the industrial process, as proven by the positive feedback from the automotive and the avionics industries. However, to be successful, static timing analysis has to face several challenges, being discussed in the subsequent Section 3.2.

3.1 Tools for Static Worst-Case Execution Time Analysis

Figure 1 shows the general structure of WCET analyzers like aiT, see `http://www.absint.com/aiT` – this is the static WCET tool we are most experienced with. The input binary executable has to undergo several analysis phases, before a worst-case execution time bound can be given for a specific task[3]. First, the binary is decoded (reconstruction of the control-flow). Next, loop and value analysis try to determine loop bounds and (abstract) contents of registers and memory cells. The (cache and) pipeline analysis computes lower and upper basic block execution time bounds. Finally, the path analysis computes the worst-case execution path through the analyzed program (see [3] for a more detailed explanation).

3.2 Challenges

A static WCET analysis has to cope with several challenges to be successful. Basically, we discern two different classes of challenges. Challenges that need to be met to make the

[3] A task (usually) corresponds to a specific entry point of the analyzed binary executable.

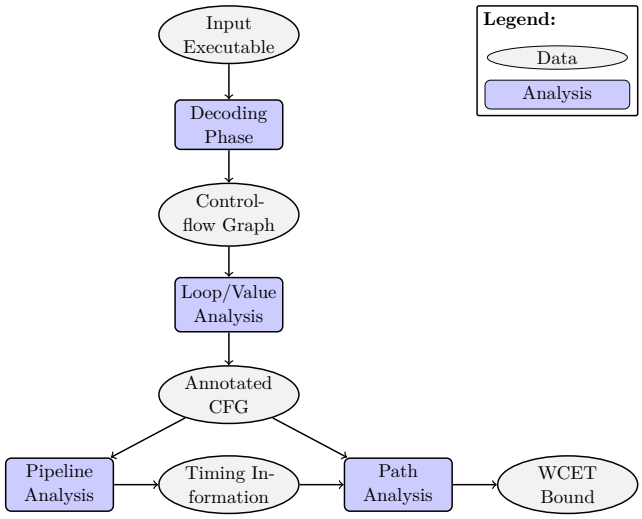


Figure 1 Phases of WCET computation.

WCET bound computation feasible at all are *tier-one* challenges. *Tier-two* challenges are concerned with keeping the WCET bounds as tight as possible, e.g., to enable a feasible schedule of the overall system.

The used coding style is tightly coupled to the encountered tier-one challenges. Section 4 investigates whether coding guidelines that are in production use (indirectly) address such challenges and whether they ease their handling. Section 4.3 provides means how to cope with tier-two challenges.

In the following, we discuss tier-one WCET analysis challenges.

Function Pointers. Often simple language constructs do not suffice to implement a certain program behavior. For instance, user-defined event handlers are usually implemented via function pointers to exchange data between communication library (e.g., for CAN devices) and the application. Resolving function pointers automatically is not easily done and sometimes not feasible at all. Nevertheless, function pointers need to be resolved to enable the reconstruction of a valid control-flow graph and the computation of a WCET bound.

Loops and Recursions. Loops (and also recursions) are a standard concept in software development. The main challenge is to automatically bound the maximum possible number of loop iterations, which is mandatory to compute a WCET bound at all. Whereas often-used counter loops can be easily bounded, it is generally infeasible to bound input-data dependent loops without additional knowledge. Similarly, such knowledge is required for recursions.

Irreducible Loops. Usually, loops have a single entry point and thus a single loop header. However, more complicated loops are occasionally encountered. By using language constructs like the `goto` statement from C or by means of hand-written assembly code, it is possible to construct loops featuring multiple entry points. So far, there exists no feasible approach to automatically bound this kind of loops [8]. Hence, additional knowledge about the control-flow behavior of such loops is always required.

4 Software Predictability

In this section we discuss existing coding standards and investigate rules from the 2004 MISRA-C standard that are beneficial for software predictability. Thereafter, we describe how design-level information can further aid static timing analysis.

4.1 Coding Guidelines

Several coding guidelines have emerged to guide software programmers to develop code that conforms to safety-critical software principles. The main goal is to produce code that does not contain errors leading to critical failure and thus causing harm to individuals or to equipment. Furthermore, software development rules aim at improved reliability, portability, and maintainability.

In 1998, the Motor Industry Software Reliability Association (MISRA) published MISRA-C [9]. The guidelines were intended for embedded automotive systems implemented in the C programming language. An updated version of the MISRA-C coding guidelines has been released in 2004 [10]. This standard is now widely accepted in other safety-critical domains, such as avionics or defense systems. On the basis of the 2004 MISRA-C standard, the Lockheed Martin Corporation has published coding guidelines that are obligatory for Air Vehicle C++ development in 2005 [2]. Albeit certain rules tackle code complexity, there are no rules that explicitly aim at developing better time predictable software.

4.2 MISRA-C

Wenzel et al. [15] reckon that among the standards DO-178B, MISRA-C, and ARINC 653, only MISRA-C includes coding rules that can effect software predictability. In the following, we thus take a closer look at the 2004 MISRA-C guidelines. The list partially corresponds to the one found in [15] (focusing on 1998 MISRA-C), but refers to the potential impact on the time predictability using binary-level static WCET analysis (e.g., with the aiT tool).

Rule 13.4 (required): ***The controlling expression of a for statement shall not contain any objects of floating type.*** State-of-the-art abstract interpretation based loop analyzers work well with integer arithmetic, but do not cope with floating point values [5, 4]. Thus, by forbidding floating point based loop conditions, a loop analysis is enabled to automatically detect loop bounds.

Rule 13.6 (required): ***Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.*** This rule promotes the use of (simple) counter-based loops and prohibits the implementation of a complex update logic of the loop counter. This allows for a less complicated loop bound detection.

Rule 14.1 (required): ***There shall be no unreachable code.*** Tools like aiT can detect that some part of the code is not reachable. However, static timing analysis computes an over-approximation of the possible control-flow. By this, the analysis might assume some execution paths that are not feasible in the actual execution of the software. Hence, the removal of unreachable parts from the code base leads to less sources of such imprecision.

Rule 14.4 (required): *The goto statement shall not be used.* The usage of the `goto` statement does not necessarily cause problems for binary-level timing analysis. These statements are compiled into unconditional branch instructions, which are no challenge to such analyses by themselves. However, the usage of the `goto` statement might possibly introduce irreducible loops into the program binary. There is no known approach available to automatically determine loop bounds for this kind of loops. Consequently, manual annotations are always required. Even worse, certain precision-enhancing analysis techniques, such as virtual loop unrolling [13], are not applicable.

Rule 14.5 (required): *The continue statement shall not be used.* Wenzel et al. [15] state that not adhering to this rule could lead to unstructured loops (see rule 14.4). However, `continue` statements only introduce additional back edges to the loop header and therefore cannot lead to irreducible loops. Any loop containing `continue` statements can be transformed into a semantically equivalent loop by means of `if-then-else` constructs. Hence, the only purpose of this rule is to enforce a certain coding style.

Rule 16.1 (required): *Functions shall not be defined with a variable number of arguments.* Functions with variable argument lists inherently lead to data dependent loops iterating over the argument list. Such loops are hard to bound automatically.

Rule 16.2 (required): *Functions shall not call themselves, either directly or indirectly.* Similarly to using `goto` statements, the use of recursive function calls might lead to irreducible loops in the call graph. Thus, a similar impact on software predictability would apply as discussed above for `goto` statements (see rule 14.4).

Rule 20.4 (required): *Dynamic heap memory allocation shall not be used.* Dynamic memory allocation leads to statically unknown memory addresses. This will lead to an over-estimation in the presence of caches or multiple memory areas with different timings. Recent work tries to address this problem by means of cache-aware memory allocation [6].

Rule 20.7 (required): *The setjmp macro and the longjmp function shall not be used.* In accordance to the discussion of rule 14.4 and of rule 16.2, the usage of the `setjmp` and the `longjmp` macro would allow the construction of irreducible loops. Hence, similar time predictability problems would arise.

4.3 Design-Level Information

Coping with all tier-one challenges of WCET analysis (see Section 3.2) is usually not sufficient in industrial practice. Additional information that is available from the design-level phase is often required to allow a computation of significantly tighter worst-case execution time bounds. Here, we address the most relevant tier-two challenges.

Operating Modes. Many embedded control software systems have different operating modes. For example, a flight control system differentiates between flight and ground mode. Any such operating mode features different functional and therefore different timing behavior. Unfortunately, the modes of behavior are not well represented in the control software code. Although there is ongoing work to semi-automatically derive operating modes from the source code [7], we still propose to methodically document their behavioral impact.

Such documentation could include loop bounds or other kinds of annotations specific to the corresponding operating mode. At best developers should instantly document the relevant source code parts to avoid a later hassle of reconstructing this particular knowledge.

Complex Algorithms. Complex algorithms or state machines are often modeled with tools like MATLAB or SCADE. By means of code generators these models are then transferred into, e.g., C code. During this process, high-level information about the algorithm or the state machine update logic respectively are lost (e.g., complex loop bounds, path exclusions).

Wilhelm et al. [16] propose systematic methods to make model information available to the WCET analyzer. The authors have successfully applied their approach and showed that tighter WCET bounds are achievable in this fashion.

Data-Dependent Algorithms. Computing tight worst-case execution time bounds is a challenging task for strongly data-dependent algorithms. This is mainly caused by two reasons. *On the one hand*, data-dependent loops are hardly bounded statically. However, for computing precise WCET bounds, it generally does not suffice to assume the maximal possible number of loop iterations for each execution context. *On the other hand*, a static analysis is often unable to exclude certain execution paths through the algorithm without further knowledge about the execution environment. The following example demonstrates this problem.

Message-based communication is usually implemented by means of fixed-size read and write buffers that are reserved for each scheduling cycle separately. During an interrupt handler the message data is either copied from or to memory – depending on the current scheduling cycle. Here, read and write operations can never occur in the same execution context of the message handler. Without further information both operations cannot be excluded by a static WCET analysis. Additionally, the analysis has no a-priori information about the amount of data being transferred. However, the allocation of the data buffers and the amount of data to transmit is statically known during the software design phase. Using this information would allow for a much more precise static timing analysis of such algorithms.

Imprecise Memory Accesses. Unknown or imprecise memory access addresses are one of the main challenges of static timing analysis for two reasons. *First*, they impair the precision of the value analysis. Any unknown read access introduces unknown values into the value analysis and therefore increases the possibly feasible control-flow paths and negatively influences the loop bound analysis. In addition, any write access to an unknown memory location destroys all known information about memory during the value analysis phase. *Second*, the pipeline analysis has to assume that any memory module might be the target of an unknown memory access – the slowest memory module will thus contribute the most to the overall WCET bound. For architectures featuring data caches, an imprecise memory access invalidates large parts of the abstract cache (or even the whole cache) and leads to an over-approximation of the possible cache misses on the WCET path. Such unknown memory accesses can result from the extensive use of pointers inside data structures with multiple levels of indirections.

A remedy to this could be to document the memory areas that might be accessed for each function separately, especially if slow memory modules could be accessed. For example, memory-mapped I/O regions that are used for CAN or FLEXRAY controllers usually are only accessed in the corresponding device driver routines. Thus, the analysis would only

Iteration Counts	Frequency of Occurrence	Observed for
0	1 552	
1	99 881 801	
2	116 421	
3	114	
4 .. 9	13	
10 .. 19	19	
20 .. 39	24	
40 .. 59	22	
60 .. 79	13	
80 .. 99	11	
100 .. 135	7	
156	1	`lDivMod(0xffd93580, 0x107d228)`
186	1	`lDivMod(0xfff2c009, 0x118dcc4)`
204	1	`lDivMod(0xffe870e3, 0x1414167)`

Table 1 Observed iteration counts for `lDivMod`.

need to assume for those specific routines that imprecise or unknown memory accesses target these (slow) memory regions. For all other routines, the analysis would be allowed to assume that different, potentially faster memory modules are being accessed.

Error Handling. In embedded software systems, error handling and recovery is a very complex procedure. In the event of an error, great care needs to be taken to ensure safety for individuals and machinery respectively.

A precise (static) timing analysis of error handling routines requires a lot more than the maximum number of possible errors that can occur or have to be handled at once. First of all however, it needs to be decided whether the error case is relevant for the worst-case behavior or not. If not, all error-case related execution paths through the software may be ignored during WCET analysis, which will obviously lead to much lower WCET bounds being computed. This however requires precise knowledge about which parts of the software are concerned with handling errors.

Otherwise, static timing analysis has to cope with error handling. The assumption that all errors might occur at once naturally leads to safe timing guarantees. However, in reality this is a rather uncommon or simply infeasible behavior of the embedded system. Here, computing tight WCET bounds requires precise knowledge about all potential error scenarios. An early documentation of the system's error handling behavior is thus expected to allow for a quicker and more precise analysis of the overall system.

Software Arithmetic. Under certain circumstances, an embedded software system makes use of software arithmetic. This is the case if the underlying hardware platform does not support the required arithmetic capabilities. For instance, the Freescale MPC5554 processor only supports single precision floating point computations [12]. If higher-precision FPU operations are required, (low-level) software algorithms emulating the required arithmetic precision come into play. Such algorithms are usually designed to provide good average-case performance, but are not implemented with good WCET predictability in mind. This often causes a static timing analysis to assume the worst-case path through such routines for most execution contexts.

An extreme example for a function with good average-case performance and bad WCET predictability is the library function `lDivMod` of the CodeWarrior V4.6 compiler for Freescale HCS12X. The purpose of this routine is to compute quotient and remainder of two 32 bit unsigned integers. The algorithm performs an iteration computing successive approximations to the final result. To get an impression on the number of loop iterations, we performed an experiment in which `lDivMod` was applied to 10^8 random inputs. Table 1 shows which iteration counts were observed in this experiment. The number of iterations is 1 in more than 99.8% and 0, 1, or 2 in more than 99.999% of the sample inputs. On the other hand, iteration counts of more than 150 could be observed for a few specific inputs. There seems to be no simple way to derive the number of iterations from given inputs (other than running the algorithm). The highest possible iteration count could not yet be determined by mathematical analysis. Even if it were known that 204 is the maximum, a worst-case execution time analysis had to assume that such a high iteration number occurs when the input values cannot be determined statically, leading to a big over-estimation of the actual WCET.

To tighten the computed WCET bounds, further information would be required to avoid the cases with high numbers of loop iterations in many or all execution contexts. Making sure that the used software arithmetic library features good WCET analyzability also helps to tighten the computed WCET bounds. Another – more radical – approach would be to employ a different hardware architecture that supports the required arithmetic precision.

5 Conclusion

Our experience with static timing analysis of embedded software systems shows that the analysis complexity varies greatly. As discussed above, the software structure strongly influences the analyzability of the overall system. Existing coding guidelines, such as the MISRA-C standard, partially address tier-one challenges encountered during WCET analysis. However, solely adhering to these guidelines does not suffice to achieve worst-case execution time bounds with the best precision possible. We usually suggest to document the software system behavior as early as possible – desirably during the software design phase – to tackle the tier-two WCET analysis challenges. Otherwise, achieving precise analysis results during the software development testing and validation phase might become a costly and time consuming process.

References

1 Armelle Bonenfant, Ian Broster, Clément Ballabriga, Guillem Bernat, Hugues Cassá, Michael Houston, Nicholas Merriam, Marianne de Michiel, Christine Rochange, and Pascal Sainrat. Coding guidelines for WCET analysis using measurement-based and static analysis techniques. Technical Report IRIT/RR–2010-8–FR, IRIT, Université Paul Sabatier, Toulouse, March 2010.

2 Lookheed Martin Corporation. C++ coding standards for the system development and demonstration program, December 2005.

3 Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile*, 807:26–42, 2010.

4 Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

5 Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Christine Rochange, editor, *Workshop on Worst-Case Execution-Time Analysis (WCET)*, volume 6 of *OASICS*, July 2007.

6 Jörg Herter, Jan Reineke, and Reinhard Wilhelm. CAMA: Cache-aware memory allocation for WCET analysis. In Marco Caccamo, editor, *Proceedings Work-In-Progress Session of the 20th Euromicro Conference on Real-Time Systems*, pages 24–27, July 2008.

7 Philipp Lucas, Oleg Parshin, and Reinhard Wilhelm. Operating mode specific WCET analysis. In Charlotte Seidner, editor, *Proceedings of JRWRTC*, October 2009.

8 Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

9 The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in vehicle based software, 1998.

10 The Motor Industry Software Reliability Association (MISRA). Guidelines for the use of the C language in critical systems, October 2004.

11 Peter Puschner and Raimund Kirner. Avoiding timing problems in real-time software. In *1st IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES 2003)*. IEEE Computer Society, 2003.

12 Freescale Semiconductor. e200z6 PowerPC Core Reference Manual, 2004.

13 Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2–3):157–179, 2000.

14 Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157–177, 2004.

15 Ingomar Wenzel, Raimund Kirner, Martin Schlager, Bernhard Rieder, and Bernhard Huber. Impact of dependable software development guidelines on timing analysis. In *Proceedings of the 2005 IEEE Eurocon Conference*, pages 575–578, Belgrad, Serbia and Montenegro, 2005. IEEE Computer Society.

16 Reinhard Wilhelm, Philipp Lucas, Oleg Parshin, Lili Tan, and Björn Wachter. Improving the precision of WCET analysis by input constraints and model-derived flow constraints. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, LNCS. Springer-Verlag, 2010. To appear.

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.PPES.2011.11

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

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 11–21

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

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.

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 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

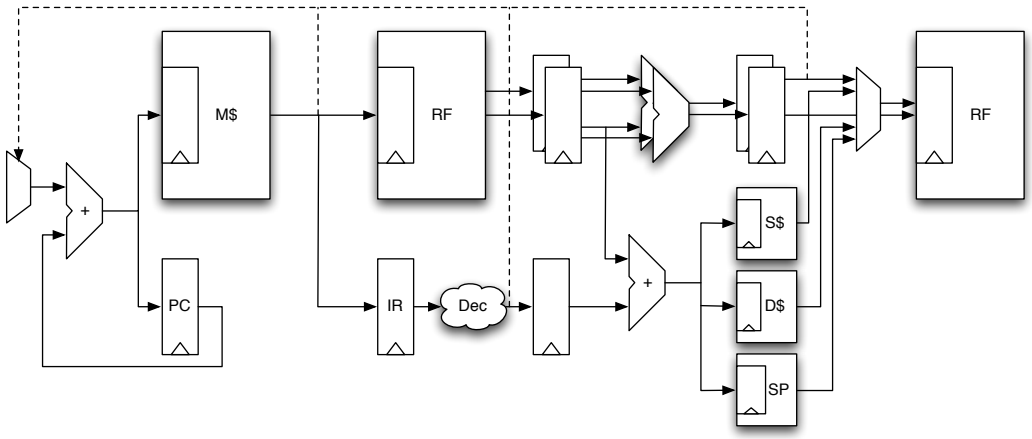


Figure 1 Pipeline of Patmos with fetch, decode, execute, and memory/write back stages.

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.

A Template for Predictability Definitions with Supporting Evidence*

Daniel Grund[1], Jan Reineke[2], and Reinhard Wilhelm[1]

1 **Saarland University, Saarbrücken, Germany.** grund@cs.uni-saarland.de
2 **University of California, Berkeley, USA.** reineke@eecs.berkeley.edu

Abstract

In real-time systems, timing behavior is as important as functional behavior. Modern architectures turn verification of timing aspects into a nightmare, due to their "unpredictability". Recently, various efforts have been undertaken to engineer more predictable architectures. Such efforts should be based on a clear understanding of predictability. We discuss key aspects of and propose a template for predictability definitions. To investigate the utility of our proposal, we examine above efforts and try to cast them as instances of our template.

Digital Object Identifier 10.4230/OASIcs.PPES.2011.22

1 Introduction

Predictability resounds throughout the embedded systems community, particularly throughout the real-time community, and has lately even made it into the Communications of the ACM [12]. The need for predictability was recognized early [25] and has since been inspected in several ways, e.g. [3, 26, 10]. Ongoing projects in point try to "reconcile efficiency and predictability" (Predator[1]), to "reintroduce timing predictability and repeatability" by extending instruction-set architectures (ISA) with control over execution time (PRET [7, 13]), or "guarantee the analyzability and predictability regarding timing" (MERASA [27]).

The common tenor of these projects and publications is that past developments in system and computer architecture design are ill-suited for the domain of real-time embedded systems. It is argued that if these trends continue, future systems will become more and more unpredictable; up to the point where sound analysis becomes infeasible — at least in its current form. Hence, research in this area can be divided into two strands: On the one hand there is the development of ever better analyses to keep up with these developments. On the other hand there is the exercise of influence on system design in order to avert the worst problems in future designs.

We do *not* want to dispute the value of these two lines of research. Far from it. However, we argue that both are often built on sand: Without a better understanding of "predictability", the first line of research might try to develop analyses for inherently unpredictable systems, and the second line of research might simplify or redesign architectural components that are in fact perfectly predictable. To the best of our knowledge there is no agreement — in the form of a formal definition — what the notion "predictability" should mean. Instead the criteria for predictability are *based on intuition* and arguments are made on a *case-by-case basis*. In the analysis of worst-case execution times (WCET) for instance, simple

* The research leading to these results has received funding from or was supported by the European Commission's Seventh Framework Programme FP7/2007-2013 under grant agreement no 216008 (Predator) and by the High-Confidence Design for Distributed Embedded Systems (HCDDES) Multidisciplinary University Research Initiative (MURI) (#FA9550-06-0312).

[1] http://www.predator-project.eu/

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 22–31

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

in-order pipelines like the ARM7 are deemed more predictable than complex out-of-order pipelines as found in the PowerPC755. Likewise static branch prediction is said to be more predictable than dynamic branch prediction. Other examples are TDMA vs. FCFS arbitration and static vs. dynamic preemptive scheduling.

The agenda of this article is to stimulate the discussion about predictability with the long-term goal of arriving at a definition of predictability. In the next section we present key aspects of predictability and therefrom derive a template for predictability definitions. In Section 3 we consider work of the last years on improving the predictability of systems and try to cast the intuitions about predictability found in these works in terms of this template. We close this section by discussing the conclusions from this exercise with an emphasis on commonalities and differences between our intuition and that of others.

2 Key Aspects of Predictability

What does predictability mean? A lookup in the Oxford English Dictionary provides the following definitions:

> predictable: adjective, able to be predicted.
> to predict: say or estimate that (a specified thing) will happen in the future or will be a consequence of something.

Consequently, a system is predictable if one can foretell facts about its future, i.e. determine interesting things about its behavior. In general, the behaviors of such a system can be described by a possibly infinite set of execution traces (sequences of states and transitions). However, a prediction will usually refer to derived properties of such traces, e.g. their length or a number of interesting events on a trace. While some properties of a system might be predictable, others might not. Hence, the first aspect of predictability is the *property to be predicted*.

Typically, the property to be determined depends on something unknown, e.g. the input of a program, and the prediction to be made should be valid for all possible cases, e.g. all admissible program inputs. Hence, the second aspect of predictability are the *sources of uncertainty* that influence the prediction quality.

Predictability will not be a boolean property in general, but should preferably offer shades of gray and thereby allow for comparing systems. How well can a property be predicted? Is system A more predictable than system B (with respect to a certain property)? The third aspect of predictability thus is a *quality measure* on the predictions.

Furthermore, predictability should be a property *inherent* to the system. Only because *some* analysis cannot predict a property for system A while it can do so for system B does not mean that system B is more predictable than system A. In fact, it might be that the analysis simply lends itself better to system B, yet better analyses do exist for system A.

With the above key aspects we can narrow down the notion of predictability as follows:

▶ **Proposition 1.** *The notion of predictability should capture if, and to what level of precision, a specified property of a system can be predicted by an optimal analysis.*

Refinements

A definition of predictability could possibly take into account more aspects and exhibit additional properties.

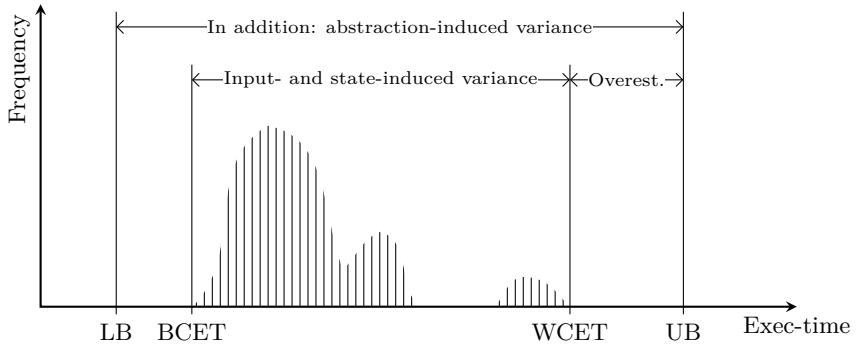


Figure 1 Distribution of execution times ranging from best-case to worst-case execution time (BCET/WCET). Sound but incomplete analyses can derive lower and upper bounds (LB, UB).

- For instance, one could refine Proposition 1 by taking into account the complexity/cost of the analysis that determines the property. However, the clause "by *any* analysis not more expensive than X" complicates matters: The key aspect of inherence requires a quantification over all analyses of a certain complexity/cost.
- Another refinement would be to consider different sources of uncertainty separately to capture only the influence of one source. We will have an example of this later.
- One could also distinguish the extent of uncertainty. E.g. is the program input completely unknown or is partial information available?
- It is desirable that the predictability of a system can be determined automatically, i.e. computed.
- It is also desirable that predictability of a system is characterized in a compositional way. This way, the predictability of a composed system could be determined by a composition of the predictabilities of its components.

2.1 A Predictability Template

Besides the key aspect of inherence, the other key aspects of predictability depend on the system under consideration. We therefore propose a template for predictability with the goal to enable a concise and uniform description of predictability instances. It consists of the above mentioned key aspects

- property to be predicted,
- sources of uncertainty, and
- quality measure.

In Section 3 we consider work of the last years on improving the predictability of systems. We then try to cast the possibly even unstated intuitions about predictability in these works in terms of this template. But first, we consider one instance of predictability in more detail to illustrate this idea.

2.2 An Illustrative Instance: Timing Predictability

In this section we illustrate the key aspects of predictability at the hand of timing predictability.

- The property to be determined is the execution time of a program assuming uninterrupted execution on a given hardware platform.

- The sources of uncertainty are the *program input* and the *hardware state* in which execution begins. Figure 1 illustrates the situation and displays important notions. Typically, the initial hardware state is completely unknown, i.e. the prediction should be valid for all possible initial hardware states. Additionally, schedulability analysis cannot handle a characterization of execution times in the form of a function depending on inputs. Hence, the prediction should also hold for all admissible program inputs.
- Usually, schedulability analysis requires a characterization of execution times in the form bounds on the execution time. Hence, a reasonable quality measure is the quotient of BCET over WCET; the smaller the difference the better.
- The inherence property is satisfied as BCET and WCET are inherent to the system.

To formally define timing predictability we need to first introduce some basic definitions.

▶ **Definition 2.** Let $\mathcal{Q}$ denote the set of all *hardware states* and let $\mathcal{I}$ denote the set of all *program inputs*. Furthermore, let $T_p(q, i)$ be the *execution time* of program p starting in hardware state $q \in \mathcal{Q}$ with input $i \in \mathcal{I}$.

Now we are ready to define timing predictability.

▶ **Definition 3** (Timing predictability)**.** Given uncertainty about the initial hardware state $Q \subseteq \mathcal{Q}$ and uncertainty about the program input $I \subseteq \mathcal{I}$, the timing predictability of a program p is

$$\Pr_p(Q, I) := \min_{q_1, q_2 \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q_1, i_1)}{T_p(q_2, i_2)} \tag{1}$$

The quantification over pairs of states in Q and pairs of inputs in I captures the uncertainty. The property to predict is the execution time T_p. The quotient is the quality measure: $\Pr_p \in [0, 1]$, where 1 means perfectly predictable.

Refinements

The above definitions allow analyses of arbitrary complexity, which might be practically infeasible. Hence, it would be desirable to only consider analyses within a certain complexity class. While it is desirable to include analysis complexity in a predictability definition it might become even more difficult to determine the predictability of a system under this constraint: To adhere to the inherence aspect of predictability however, it is necessary to consider *all* analyses of a certain complexity/cost.

Another refinement is to distinguish hardware- and software-related causes of unpredictability by separately considering the sources of uncertainty:

▶ **Definition 4** (State-induced timing predictability)**.**

$$\mathrm{SIPr}_p(Q, I) := \min_{q_1, q_2 \in Q} \min_{i \in I} \frac{T_p(q_1, i)}{T_p(q_2, i)} \tag{2}$$

Here, the quantification expresses the maximal variance in execution time due to different hardware states, q_1 and q_2, for an arbitrary but fixed program input, i. It therefore captures the influence of the hardware, only. The input-induced timing predictability is defined analogously. As a program might perform very different actions for different inputs, this captures the influence of software:

▶ **Definition 5** (Input-induced timing predictability)**.**

$$\mathrm{IIPr}_p(Q, I) := \min_{q \in Q} \min_{i_1, i_2 \in I} \frac{T_p(q, i_1)}{T_p(q, i_2)} \tag{3}$$

Example for state-induced timing unpredictability

A system exhibits a *domino effect* [14] if there are two hardware states q_1, q_2 such that the difference in execution time of the same program starting in q_1 respectively q_2 may be arbitrarily high, i.e. cannot be bounded by a *constant*. For instance, the iterations of a program loop never converge to the same hardware state and the difference in execution time increases in each iteration.

In [22] Schneider describes a domino effect in the pipeline of the PowerPC 755. It involves the two asymmetrical integer execution units, a greedy instruction dispatcher, and an instruction sequence with read-after-write dependencies.

The dependencies in the instruction sequence are such that the decisions of the dispatcher result in a longer execution time if the initial state of the pipeline is empty than in case it is partially filled. This can be repeated arbitrarily often, as the pipeline states after the execution of the sequence are equivalent to the initial pipeline states. For n subsequent executions of the sequence, execution takes $9n+1$ cycles when starting in one state, q_1^*, and $12n$ cycles when starting in the other state, q_2^*. Hence, the state-induced predictability can be bounded for such programs p_n:

$$\mathrm{SIPr}_{p_n}(Q, I) = \min_{q_1, q_2 \in Q} \min_{i \in I} \frac{T_{p_n}(q_1, i)}{T_{p_n}(q_2, i)} \leq \frac{T_{p_n}(q_1^*, i^*)}{T_{p_n}(q_2^*, i^*)} = \frac{9n+1}{12n} \tag{4}$$

3 Supporting Evidence?

In recent years, significant efforts have been undertaken to design more predictable architectural components. As we mentioned in the introduction, these efforts are usually based on sensible, yet informal intuitions of what makes a system predictable. In this section, we try to cast these intuitions as instances of the predictability template introduced in Section 2.1.

We summarize our findings about how existing efforts fit into our predictability template in Tables 1 and 2. For each approach we determine the *property* it is concerned with, e.g. execution time, the *source of uncertainty* that makes this property unpredictable, e.g. uncertainty about program inputs, and the *quality measure* that the approach tries to improve, e.g. the variation in execution time. Whenever the goals that are explicitly stated in the referenced papers do not fit into this scheme, we determine whether the approach can still be explained within the scheme. In that case, we provide appropriate characterizations in parentheses. In the following sections, we supplement the two tables with brief descriptions of the approaches.

3.1 Branch Prediction

Bodin and Puaut [5] and Burguière and Rochange [6] propose WCET-oriented static branch prediction schemes. Bodin and Puaut specifically try to minimize the number of branch mispredictions along the worst-case execution path, thereby minimizing the WCET. Using static branch prediction rather than dynamic prediction is motivated by the difficulty in modeling complex dynamic schemes and by the incurred analysis complexity during WCET estimation. The approaches are evaluated by comparing WCET estimates for the generated static predictions with WCET estimates for the dynamic scheme, based on conservative approximations of the number of mispredictions.

Table 1 Part I of constructive approaches to predictability.

Approach	Hardware unit(s)	Property	Source of uncertainty	Quality measure
WCET-oriented static branch prediction [5, 6]	Branch predictor	Number of branch mispredictions	Analysis imprecision (Uncertainty about initial predictor state)	Statically computed bound (Variability in mispredictions)
Time-predictable execution mode for superscalar pipelines [21]	Superscalar out-of-order pipeline	Execution time of basic blocks	Analysis imprecision (Uncertainty about the pipeline state at basic block boundaries)	Qualitative: analysis practically feasible (Variability in execution times of basic blocks)
Time-predictable Simultaneous Multithreading [2, 16]	SMT processor	Execution time of tasks in real-time thread	Uncertainty about execution context, i.e., other tasks executing in non-real-time threads	Variability in execution times
CoMPSoC: a template for composable and predictable multiprocessor system on chips [9]	System on chip including network on chip, VLIW cores and SRAM	Memory access and communication latency	Concurrent execution of unknown other applications	Variability in latencies
Precision-Timed Architectures [13]	Thread-interleaved pipeline and scratchpad memories	Execution time	Uncertainty about initial state and execution context	Variability in execution times
Predictable out-of-order execution using virtual traces [28]	Superscalar out-of-order pipeline and scratchpad memories	Execution time of program paths	State of features such as caches, branch predictors, etc. and input values of variable latency instructions	Variability in execution times
Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems [29]	Pipeline, memory hierarchy, and buses	Execution time, memory access latencies, latencies of bus transfers	Uncertainty about the pipeline state, the cache state, and about concurrently executing applications	Variability in execution times and memory access latencies

3.2 Pipelining and Multithreading

Rochange and Sainrat [21] propose a time-predictable execution mode for superscalar pipelines. They simplify WCET analysis by regulating instruction flow of the pipeline at the beginning of each basic block. This removes all timing dependencies within the pipeline between basic blocks. Thereby it reduces the complexity of WCET analysis, as it can be performed on each basic block in isolation. Still, caches have to be accounted for globally. The authors take the stance that efficient analysis techniques are a prerequisite for predictability: "a processor might be declared unpredictable if computation and/or memory requirements to analyse the WCET are prohibitive."

Barre et al. [2] and Mische et al. [16] propose modifications to simultaneous multithreading (SMT) architectures. They adapt thread-scheduling in such a way that one thread, the real-time thread, is given priority over all other threads, the non-real-time threads. As a consequence, the real-time thread experiences no interference by other threads and can be analyzed without having to consider its context, i.e., the non-real-time threads.

3.3 Comprehensive Approaches

Hansson et al. [9] propose CoMPSoC, a template for multiprocessors with predictable and composable timing. By predictability they refer to the ability to determine lower bounds on performance. By composability they mean that the composition of applications on one platform does not have any influence on their timing behavior. Predictability is achieved by VLIW cores and no use of caches or DRAM. Composability is achieved by TDM arbitration on the network on chip and on accesses to SRAMs.

Lickly et al. [13] present a precision-timed (PRET) architecture that uses a thread-interleaved pipeline and scratchpad memories. The thread-interleaved pipeline provides high overall throughput and constant execution times of instructions in all threads, at the sacrifice of single-thread performance. PRET introduces new instructions into the ISA to provide control over timing at the program level.

Whitham and Audsley [28] refine the approach of Rochange [21]. Any aspect of the pipeline that might introduce variability in timing is either constrained or eliminated: scratchpads are used instead of caches, dynamic branch prediction is eliminated, variable duration instructions are modified to execute a constant number of cycles, exceptions are ignored. Programs are statically partitioned into so-called traces. Within a trace, branches are predicted perfectly. Whenever a trace is entered or left, the pipeline state is reset to eliminate any influence of the past.

Wilhelm et al. [29] give recommendations for future architectures in time-critical embedded systems. Based on the principle *to reduce the interference on shared resource*, they recommend to use caches with LRU replacement, separate instruction and data caches, and so-called *compositional* architectures, such as the ARM7. Such architectures do not have domino effects and exhibit little state-induced variation in execution time.

3.4 Memory Hierarchy

In the context of the Java Optimized Processor, Schoeberl [23] introduces the so-called *method cache*: instead of caching fixed-size memory blocks, the method cache caches entire Java methods. Using the method cache, cache misses may only occur at method calls and returns. Due to caching variable-sized blocks, LRU replacement is infeasible. Metzlaff et al. [15] propose a very similar structure, called *function scratchpad*, which they employ within an SMT processor.

Schoeberl et al. [24] propose dedicated caches for different types of data: methods (instructions), static data, constant, stack data, and heap data. For heap data, they propose a small, fully-associative cache. Often, the addresses of accesses to heap data are difficult, or in case of most memory allocators, impossible to predict statically. In a normal set-associative cache, an access with an unknown address may modify any cache set. In the fully-associative case, knowledge of precise memory addresses for heap data is unnecessary.

Puaut and Decotigny [18] propose to statically lock cache contents to eliminate intra-

Table 2 Part II of constructive approaches to predictability.

Approach	Hardware unit(s)	Property	Source of uncertainty	Quality measure
Method Cache [23, 15]	Memory hierarchy	Memory access time	(Uncertainty about initial cache state)	Simplicity of analysis
Split Caches [24]	Memory hierarchy	Number of data cache hits	Among others, uncertainty about addresses of data accesses	(Percentage of accesses that can be statically classified)
Static Cache Locking [18]	Memory hierarchy	Number of instruction cache hits	Uncertainty about initial cache state and interference due to preempting tasks	Statically computed bound (Variability in number of hits)
Predictable DRAM Controllers [1, 17]	DRAM controller in multi-core system	Latency of DRAM accesses	Occurrence of refreshes and interference by concurrently executing applications	Existence and size of bound on access latency
Predictable DRAM Refreshes [4]	DRAM controller	Latency of DRAM accesses	Occurrence of refreshes	Variability in latencies
Single-path paradigm [19]	Software-based	Execution time	Uncertainty about program inputs	Variability in execution times

task cache interference and inter-task cache interferences (in preemptive systems). They introduce two low-complexity algorithms to statically determine which instructions to lock in the cache. To evaluate their approach, they compare statically guaranteed cache hit rates in unlocked caches with hit rates in locked caches.

Akesson et al. [1] and later Paolieri et al. [17] propose the predictable DRAM controllers *Predator* and *AMC*, respectively. These controllers provide a guaranteed maximum latency and minimum bandwidth to each client, independently of the execution behavior of other clients. This is achieved by predictable access schemes, which allow to bound the latencies of individual memory requests, and predictable arbitration mechanisms: CCSP in *Predator* and TDM in *AMC*, allow to bound the interference between different clients.

Bhat and Mueller [4] eliminate interferences between DRAM refreshes and memory accesses, so that WCET analysis can be performed without considering refreshes. Standard memory controllers periodically refresh consecutive rows. Their idea is to instead execute these refreshes in bursts and refresh all lines of a DRAM device in a single or few bursts. Such refresh bursts can then be scheduled in periodic tasks and taken into account during schedulability analysis.

3.5 Discussion

The predictability view of most efforts can indeed be cast as instances of the predictability template introduced in Section 2.1. Also, different efforts *do* require different instantiations: Properties found include: execution time, number of branch mispredictions, number of cache misses, DRAM access latency. Sources of uncertainty include: initial {processor|cache|branch predictor} state, but also program inputs, and concurrently executing applications. Most disagreement between the predictability template and the views taken in the analyzed efforts arises at the question of the quality measure: Many approaches use existing static analysis approaches to evaluate the predictability improvement. This *does not* establish that an approach improves predictability. However, as the inherent predictability is often hard to determine, this is still useful. Designers of real-time systems need analysis methods that will provide useful guarantees. So, from a practical point of view, system A *will* be considered more predictable than system B if some analysis for A are more precise than for B. In such cases, further research efforts should clarify whether A is indeed more predictable than B. Overapproximating static analyses provide upper bounds on a system's inherent predictability. Few methods exist so far to bound predictability from below.

4 Related Work

Here we want to discuss related work that tries to capture the essence of predictability or aims at a formal definition.

Bernardes [3] considers a discrete dynamical system (X, f), where X is a metric space and f describes the behavior of the system. Such a system is considered predictable at a point a, if a predicted behavior is sufficiently close to the actual behavior. The actual behavior at a is the sequence $(f^i(a))_{i\in\mathbb{N}}$ and the predicted behavior is a sequence of points in δ-environments, $(a_i)_{i\in\mathbb{N}}$, where $a_i \in B(f(a_{i-1}), \delta)$, and the sequence starts at $a_0 \in B(a, \delta)$.

Stankovic and Ramamritham [25] already posed the question about the meaning of predictability in 1990. The main answers given in this editorial is that "it should be possible to show, demonstrate, or prove that requirements are met subject to any assumptions made." Hence, it is rather seen as the existence of successful analysis methods than an inherent system property.

Henzinger [10] describes predictability as a form of determinism. Several forms of nondeterminism are discussed. Only one of them influences observable system behavior, and thereby qualifies as a source of uncertainty in our sense. There is also a short discussion how to deal with such nondeterminism: Either avoid it by building systems bottom-up using only deterministic components or achieve top-level determinism by hiding lower-level nondeterminism by a deterministic abstraction layer. [25] discusses a similar approach.

Thiele and Wilhelm [26] describe threats to timing predictability of systems, and proposes design principles that support timing predictability. Timing predictability is measured as difference between the worst (best) case execution time and the upper (lower) bound as determined by an analysis.

In a precursor of this article, Grund [8] also attempts to formally capture predictability. It is argued, as opposed to almost all prior attempts, that predictability should be an inherent system property.

Kirner and Puschner [11] describe time-predictability as the ability to calculate the duration of actions and explicitly includes the availability of efficient calculation techniques. Furthermore, a "holistic definition of time-predictability" is given. It combines the predictability of timing, as given in [8] and in Equation 1; and the predictability of the worst-case timing, as given in [26].

[20] does not aim at a general definition of predictability. Instead the predictability of caches, in particular replacement policies, is considered. Two metrics are defined that indicate how quickly uncertainty, which prevents the classification of hits respectively misses, can be eliminated. As these metrics mark a limit on the precision that *any* cache analysis can achieve, they are inherent system properties.

5 Summary and Future Work

The most severe disagreement between our opinion on predictability and those of others concerns the inherence property. We think that the aspect of inherence is indispensable to predictability: Basing the predictability *of a system* on the result of *some* analysis of the system is like stating that sorting takes exponential time only because nobody has found a polynomial algorithm yet!

Modern computer architectures are so complex that arguing about properties of their timing behavior as a whole is extremely difficult. We are in search of compositional notions of predictability, which would allow us to derive the predictability of such an architecture from that of its pipeline, branch predictor, memory hierarchy, and other components. Future work should also investigate the relation of predictability to other properties such as robustness, composability and compositionality.

References

1 B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *CODES+ISSS '07*, pages 251–256, 2007.

2 J. Barre, C. Rochange, and P. Sainrat. A predictable simultaneous multithreading scheme for hard real-time. In *Architecture of computing systems '08*, pages 161–172, 2008.

3 N. C. Bernardes, Jr. On the predictability of discrete dynamical systems. *Proc. of the American Math. Soc.*, 130(7):1983–1992, 2001.

4 B. Bhat and F. Mueller. Making DRAM refresh predictable. In *ECRTS '10*, 2010.

5 F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. In *ECRTS '05*, pages 33–40, 2005.

6 C. Burguiere, C. Rochange, and P. Sainrat. A case for static branch prediction in real-time systems. In *RTCSA '05*, pages 33–38, 2005.

7 S. Edwards and E. Lee. The case for the precision timed (PRET) machine. In *DAC '07*, pages 264–265, 2007.

8 D. Grund. Towards a formal definition of timing predictability. Presentation at RePP 2009 workshop. `http://rw4.cs.uni-saarland.de/~grund/talks/repp09-preddef.pdf`.

9 A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *Trans. Des. Autom. Electron. Syst.*, 14(1):1–24, 2009.

10 T. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philos. Trans. Royal Soc.: Math., Phys. and Engin. Sciences*, 366(1881):3727–3736, 2008.

11 R. Kirner and P. Puschner. Time-predictable computing. In *SEUS '11*, volume 6399 of *LNCS*, pages 23–34, 2011.

12 Edward Lee. Computing needs time. *Comm. of the ACM*, 52(5):70–79, 2009.

13 B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable programming on a precision timed architecture. In *CASES '08*, pages 137–146, 2008.

14 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '09*, pages 12–21, 1999.

15 S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *MEDEA '08*, pages 38–45, 2008.

16 J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting spare resources of in-order SMT processors executing hard real-time threads. In *ICCD '08*, pages 371–376, 2008.

17 M. Paolieri, E. Quinones, F.J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Syst. Letters*, 1(4):86–90, 2009.

18 I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02*, page 114, 2002.

19 P. Puschner and A. Burns. Writing temporally predictable code. In *WORDS '02*, page 85, 2002.

20 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

21 C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *Computing Frontiers '05*, pages 307–314, 2005.

22 J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

23 M. Schoeberl. A time predictable instruction cache for a Java processor. In *JTRES '04*, pages 371–382, 2004.

24 M. Schoeberl, W. Puffitsch, and B. Huber. Towards time-predictable data caches for chip-multiprocessors. In *SEUS '09*, pages 180–191, 2009.

25 J. Stankovic and K. Ramamritham. What is predictability for real-time systems? *Real-Time Syst.*, 2:247–254, 1990.

26 L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3):157–177, 2004.

27 T. Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99, 2010.

28 J. Whitham and N. Audsley. Predictable out-of-order execution using virtual traces. In *RTSS '08*, pages 445–455, 2008.

29 R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. on CAD of Integrated Circuits and Syst.*, 28(7):966–978, 2009.

An Overview of Approaches Towards the Timing Analysability of Parallel Architectures

Christine Rochange[1]

1 **Institut de Recherche en Informatique de Toulouse**
University of Toulouse, France
rochange@irit.fr

Abstract

In order to meet performance/low energy/integration requirements, parallel architectures (multithreaded cores and multi-cores) are more and more considered in the design of embedded systems running critical software. The objective is to run several applications concurrently. When applications have strict real-time constraints, two questions arise: a) how can the worst-case execution time (WCET) of each application be computed while concurrent applications might interfere? b) how can the tasks be scheduled so that they are guarantee to meet their deadlines? The second question has received much attention for several years [4, 8]. Proposed schemes generally assume that the first question has been solved, and in addition that they do not impact the WCETs. In effect, the first question is far from been answered even if several approaches have been proposed in the literature. In this paper, we present an overview of these approaches from the point of view of static WCET analysis techniques.

1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

Keywords and phrases WCET analysis, multicore, time predictability

Digital Object Identifier 10.4230/OASIcs.PPES.2011.32

1 Introduction

Parallel architectures, including multithreaded processors (MT) and multi-cores (MC), are being increasingly used in embedded systems because they fulfill various requirements like high performance, reduced energy consumption and thermal dissipation, and high integration. This is achieved through resource sharing among tasks: space sharing in instruction queues (MT) or caches (MT&MC), and time sharing in the pipeline (MT) or on the shared bus to the memory hierarchy (MC).

Now, in hard real-time systems, some tasks have strict deadlines and they must be carefully scheduled to meet them. Task scheduling algorithms rely on the knowledge of the WCET of each task. Research on timing analysis has been carried out for more than fifteen years. The proposed approaches range from testing techniques, that estimate the worst-case execution time from observed execution times (either on the target hardware or on a cycle-accurate simulator) which is clearly unsafe for critical software, to solutions based on static software analysis techniques that compute safe WCETs provided the model of the target hardware is correct. In this paper, we focus on static WCET analysis which is the most appropriate when considering hard real-time tasks but also the most sensible to non deterministic instructions timings.

Until recently, static WCET analysis has assumed that the task under analysis could not be impacted by any external event (either related to another task or to hardware-level

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 32–41

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

devices like timer interrupts or memory refreshes). Unfortunately, resource sharing in a parallel architecture questions this assumption since it induces tasks interferences that are likely to impact instructions timings. Such interferences include conflicts to access a shared resource, which are solved by stalling all the requesting tasks but one, as well as corruption in memories when a task invalidates part of the contents that was used by another task.

Recent work has focused on these issues and different kinds of approaches have been proposed: some intend to take the possible interferences into account when computing the WCET of a task, others aim at controlling the interactions to make the WCET analysis easier. In the latter category, some solutions require detailed knowledge of all the tasks that may execute concurrently to the task to be analyzed, while other solutions make it possible to determine the WCET without knowing anything about the concurrent tasks. In this paper, we review all these approaches and we discuss their relevance from the point of view of static WCET analysis.

The paper is organized as follows. Section 2 gives a short overview of static WCET analysis techniques with special focus on hardware-specific parts and shows how resource sharing may impact instruction timings. A general overview of the approaches that have been proposed to deal with inter-task interferences is given in Section 3. In Sections 4 and 5, techniques related to handling storage and bandwidth resource sharing respectively are presented. Concluding remarks are given in Section 6.

2 Static WCET analysis and impact of resource sharing

2.1 Static WCET analysis

Techniques for static WCET analysis have been investigated for the last fifteen years. The proposed solutions rely on a number of assumptions: the WCET is computed for a task considered alone, that is not impacted by any other task or external event, that cannot be preempted by the system scheduler (except for specific works on the effects of preemptions, like [3]) and that cannot be interrupted.

Static WCET analysis typically requires three steps. The flow analysis builds the Control Flow Graph of the application from its executable code, and determines flow facts like loop bounds and infeasible paths from the source code [10, 15, 21]. The low-level analysis computes the worst-case execution costs of basic blocks taking into account the specifications of the target hardware and will be detailed below. Finally the WCET computation combines the flow facts and the execution costs to find out the longest path and its execution time: one popular method for this computation is the Implicit Path Enumeration Technique (IPET) [17] based on integer linear programming techniques.

The low-level analysis step breaks down into two sub-steps. The first one examines the behavior of history-based components, mainly the instruction and data caches: the most popular approaches are based on abstract interpretation techniques [6] and assign a category to each access to the cache (`ALWAYS_MISS`, `ALWAYS_HIT`, `PERSISTENT` or `NOT_CLASSIFIED`). Existing solutions consider set-associative instruction and data caches [11], or multi-level cache hierarchies [13]. The second part of low-level analysis computes the execution cost of each basic block when executed in the pipeline [34, 18, 32]. When examining the way a basic block is processed through the pipeline, any possible context (initial pipeline state) must be considered. The existing algorithms differ in how this context is expressed: as a worst-case pipeline state [18], as an abstract state built by abstract interpretation [34] or as a set of parameters that represent the availability of every pipeline resource [32]. But they are in

agreement on the fact that they derive block costs from relative (instead of absolute) start and finish times. The impact of the cache latencies (related to the previously determined categories) may be taken into account when estimating the block costs or considered globally in the WCET computation step (which is likely to be less precise, and even unsafe for processors that make timing anomalies [20, 31] possible).

2.2 Impact of resource sharing on instructions timings

Simultaneous multithreading (SMT) processors execute several threads concurrently to improve the usage of hardware resources (mainly functional units) [38]. Common resources (instruction queues, functional units, but also instruction and data caches and branch predictor tables) are shared between concurrent threads. Some of these resources (instruction queues and buffers, caches) are referred to as *storage* resources because they keep information for a while, generally for several cycles. On the contrary, *bandwidth* resources (e.g. functional units or commit stage) are typically reallocated at each cycle [5]. A similar terminology can be used for the shared resources in a multicore architecture: a cache that is shared among the cores is a storage resource while a common bus to the memory hierarchy is a bandwidth resource.

Resource sharing is likely to impact the instructions timing. For a bandwidth resource, possible conflicts between concurrent threads to access the resource may delay some of the threads. As a result, some instruction latencies are lengthened. In an SMT core, delayed instructions may spend more time than expected in some of the pipeline stages. In a multicore, the latency of an access to the main memory may be increased because of the waiting time to the bus.
The effects of sharing storage resources are two-fold. On the one hand, the resource capacity that is usable by a thread may be less than expected since some entries may be occupied by other threads. In an SMT core, this may result in instructions being stalled in a pipeline stage because their destination queue is full. On the other hand, shared memories like caches or branch predictor tables may have their contents corrupted by other threads which could produce either destructive or constructive effects. A destructive effect is observed when another thread degrades the memory contents from the point of view of the thread under analysis: for example, another thread replaces a cache line that had been loaded by the analyzed thread and is still useful. On the contrary, a constructive effect improves the situation for the thread under analysis: for example, a cache line that it requires has been brought into the cache by another thread (this may happen when the threads share parts of code or data). However, even what is seen as constructive in the average case might impair the results of WCET analysis if the processor suffers from timing anomalies [20, 31] (in that case, a miss in the cache does not always lead to the worst-case execution time).

It is absolutely *unsafe* to ignore the effects of resource sharing when computing WCETs. Although we focus on static WCET analysis throughout this paper, we also insist that it is at least equally unsafe to rely on measurement-based timing analysis on a parallel architecture since it is very unlikely that all the possible threads interferences can be observed. In the next section, we review various approaches that have been investigated to cope with these difficulties.

3 General approaches to WCET analysis/analysability of concurrent applications

We have found three kinds of approaches to the problem of accounting for parallel tasks interferences when computing the WCET of one of these tasks. They differ from each other by the way they consider that the impact of concurrent tasks should be taken into account. In the following, τ represents a task under WCET analysis while T stands for the set of its concurrent tasks.

In this section, we give the main principles of these approaches. How they have been instantiated in the literature is described later in the paper.

3.1 Joint WCET analysis of tasks

A first category of approaches to the WCET analysis of a task executed in parallel to other tasks includes the solutions that consider the set of tasks altogether in order to determine their possible interactions. As far as storage resources are concerned, this means analyzing the code of each task in $T \cup \{\tau\}$ to determine possible conflicts, and then accounting for the impact of these conflicts on τ's WCET. For bandwidth resources, identifying conflicts generally requires considering all the possible task interleavings which is likely to be complex with fine-grained interleavings (e.g. at instruction- or memory access-level).

The feasibility of joint analysis techniques relies on all the co-running tasks being known at analysis time. This might be an issue when considering a mixed-criticality workload for which non critical tasks are dynamically scheduled (then any non critical tasks in the system should be considered as a potential opponent). In addition, it may happen that the non critical tasks have not been developed with WCET-analysis in mind and they may not be analyzable, e.g. due to tricky control flow patterns. Also, even with an homogeneously critical workload, the set of tasks that may be co-scheduled with the task under analysis depends on the schedule which, in turn, is determined from the tasks WCETs. This issue might be tackled through an iterative process but we are not aware of any work on this topic.

3.2 Statically-controlled resource sharing

Acknowledging the difficulty of analyzing storage and bandwidth conflicts accurately, a number of solutions have been proposed to statically master the task interferences so that they might be more easily taken into account in the WCET analysis. The techniques in this category all require having knowledge of the complete workload.

Controlling interferences in storage resources generally consists in limiting such interferences by restricting accesses to the shared resource. As we will see in the next sections, the proposed techniques of this kind really tend to meet the requirements of static WCET analysis techniques in terms of reduced complexity, but the solutions basic on static control proposed for bandwidth resources do not fit the principles of static WCET analysis.

3.3 Task isolation techniques

The third category of approaches includes all those that intend to make it possible to analyze the WCET of a task/thread without any knowledge about the concurrent tasks/threads. This is achieved through the design of hardware schemes that exhibit predictable behavior for shared resources. For storage resources, a common approach is to partition the storage among the tasks, so that each critical task has a private partition. For bandwidth resources,

an appropriate arbitration is needed, that guarantees upper bound delays independently of the workload.

In the following, we review the techniques that have been proposed so far and that belong to these three categories.

4 Approaches to analyze storage resource sharing

4.1 Joint analysis of memories

Several recent papers focus on the analysis of the possible corruption of L2 shared instruction caches by concurrent tasks [40, 41, 12]. The general process is the following: L1 and L2 instruction cache analysis is first performed for each task in $T \cup \{\tau\}$ independently, ignoring interferences, using usual techniques [11]; then the results of the analysis of the L2 cache for task τ are modified considering that each cache set used by another task in T is likely to be corrupted. For a direct-mapped cache, as studied by Yan and Zhang [40], any access to a conflicting set is classified as `ALWAYS_MISS` (should be `NOT_CLASSIFIED` if timing anomalies may occur). For a set-associative cache, as considered by Li *et al.* [41] and Hardy *et al.* [12], possible conflicts impact the ages of cache lines.

The main concern with this general approach is its scalability to large tasks: if the number of possible concurrent tasks is large and if these tasks span widely over the L2 cache, we expect most of the L2 accesses to be `NOT_CLASSIFIED` which may lead to an overwhelmingly overestimated WCET. For this reason, Li *et al.* [41] refine the technique by introducing an analysis of tasks lifetimes, so that tasks that cannot be executed concurrently (according to the scheduling algorithm, which is non-preemptive and static priority-driven in this paper, and to inter-tasks dependencies) are not considered as possibly conflicting. Their framework involves an iterative worst-case response time analysis process, where each iteration (i) estimates the BCET and WCET of each task according to expected conflicts in the L2 cache; (ii) determines the possible tasks schedules, which may show that some tasks cannot overlap (the initial assumption is that all tasks overlap). This approach is likely to reduce pessimism but may not fit independent tasks with a more complex scheduling scheme. Another solution to the complexity issue has been proposed by Hardy *et al.* [12]: they introduce a compiler-directed scheme that enforces L2 cache bypassing for single-usage program blocks. This sensibly reduces the number of possible conflicts. Lesage *et al.* [16] have recently extended this scheme to shared data caches.

4.2 Storage partitioning and locking schemes

Cache partitioning and locking techniques have first been proposed as a means to simplify the cache behavior analysis in single-core non-preemptive systems [27, 26, 30, 25]. Recently, these techniques have been investigated by Suhendra and Mitra [37] to assess their usability in the context of shared caches in multicore architectures. They consider combinations of (static or dynamic) locking schemes and (core-based or task-based) partitioning techniques. They find out that (i) core-based partitioning strategies (where each core has a private partition and any task can use the entire partition of the core it is running on) outperform task-based algorithms; (ii) dynamic locking techniques, that allow reloading the cache during execution, lead to lower WCETs than static approaches.

Paolieri *et al.* [23] investigate software-controlled hardware cache partitioning schemes. They consider columnization (each core has a private write access to one or several ways in a set-associative cache) and bankization (each core has a private access to one or several cache

banks) techniques. In both cases, the number of ways/banks allocated to each core can be changed by software, but it is assumed to be fixed all along the execution of a given task. They show that bankization leads to tighter WCET estimates.

Techniques to achieve timing-predictability in SMT processors are also based on partitioning instructions queues [1, 22].

5 Approaches to analyze bandwidth resources sharing

5.1 Joint analysis of conflict delays

Crowley and Baer have considered the case of a network processor running pipelined packet handling software [7]. The application includes several threads, each one implementing one stage of the computation. The processor features fine-grained multithreading: it provides specific hardware to store the architectural state of several threads, which allows fast context switching, and switches to another thread whenever the current thread is stalled on a long-latency operation. The time during which a thread is suspended depends on the time the other threads can execute before, in turn, yielding control so that the first thread can resume its execution. The proposed approach consists in determining the overall WCET of the application (set of concurrent threads) by considering the threads altogether. The Control Flow Graphs used for static WCET analysis are augmented with *yield nodes* at the points where the threads will yield control. *Yield edges* link each yield node of a given thread to all the return-from-yield nodes of any other thread that is likely to be selected when it is suspended. This results in a complex global Control Flow Graph which, in addition to the control flow of each thread, expresses the possible control flow from one thread to another. From this CFG, an integer linear program is built and used to determine the overall WCET of the application, using the IPET method [17]. Our feeling is that such an approach is not scalable and cannot handle complex applications.

5.2 Statically-scheduled access to shared bandwidth resources

To improve the analysability of latencies to a shared bus in a multicore architecture, Rosén *et al.* [33] introduce a TDMA-based bus arbiter. A *bus schedule* contains a number of slots, each allocated to one core, and is stored in a table in the hardware. At run-time, the arbiter periodically repeats the schedule and grants the bus to the core the current slot has been assigned to. The idea behind this scheme is that a predefined bus schedule makes the latencies of bus accesses predictable for WCET analysis. This relies on the assumption that it is possible, during the low level analysis, to determine the start time of each node (basic block) in the CFG so that it can be decided whether an access to the bus is within a bus slot allocated to the core or is to be delayed. This assumption does not hold for static WCET analysis techniques. It would require unrolling all the possible paths in the CFG which clearly goes against the root principles of static analysis. Moreover, in the case of multiple possible paths (which is the common case), a block is likely to exhibit a large number of possible start times which will noticeably complicate the WCET computation. Alternatively, the delay to get access to the bus could be upper bounded by the sum of the other slots lengths. This would come to the simple round-robin solution discussed below if slots are as short as the bus latency, but would probably severely degrade the worst-case performance with longer slots. For these reasons, we believe that static WCET analysis can get advantage of static bus scheduling only for applications that exhibit a very limited number of execution paths, as targeted by the single-path programming paradigm [28].

5.3 Task-independent bandwidth partitioning schemes

Solutions to make the latencies to shared bandwidth resources predictable reside in bandwidth partitioning techniques. This is what we call *task isolation*: an upper bound of the shared resource latency is known (it does not depend on the nature of the concurrent tasks) and can be considered for WCET analysis.

Mische *et al.* [22] introduce CarCore, a multithreaded embedded processor that supports one hard real-time thread (HRT) together with non critical threads. Temporal thread isolation is ensured for the HRT only, in such a way that its WCET can be computed as if it was executed alone in the processor (i.e. its execution time cannot be impacted by any other thread).

When considering multiple critical threads running simultaneously either in an SMT core or in a multi-core architecture (with one hard real-time thread per core), most of the approaches are based on Round-Robin-like arbitration which allows considering an upper bound on the latency to the shared resource: $D = N \times L - 1$ where L is the latency of the resource and N is the number of competing tasks. Barre *et al.* [1] propose an architecture for an SMT core supporting several critical threads: to provide time-predictability, the storage resources (e.g. instruction queues) are partitioned and the bandwidth resources (e.g. functional units) are scheduled by such a round-robin scheme. Paolieri *et al.* [23] propose a round-robin-like bus arbiter to the shared memory hierarchy in a multi-core architecture. This scheme is completed by a time-predictable memory controller [24] that also guarantees upper bounds on the main memory latencies. Bourgade *et al.* [2] introduce a multiple-bandwidth bus arbiter where each core is assigned a priority-level that defines its upper-bound delay to get access to the bus. This scheme better fits workloads where threads exhibit heterogeneous demands to the main memory.

The MERASA project [39] funded by the European Community (FP7 program) has designed a complete time-predictable multicore architecture with SMT cores, that implements some of the mechanisms mentioned above.

The PRET architecture [19] is built around a thread-interleaved pipeline: it includes private storage resources for six threads and each of the six pipeline stages processes an instruction from a different thread. To prevent long-latency instructions from stalling the pipeline and thus impacting the other threads, these instructions are replayed during the thread's slots until completion. Each thread has private instruction and data scratchpad memories and the off-chip memory is accessed through a *memory wheel* scheme where each thread has its own access window.

6 Conclusion

Parallel architectures are more and more frequently used in embedded system designs. However, they raise timing-analysability issues for critical applications for which worst-case execution time must be computed. Recent research on WCET analysis techniques and real-time systems design address this topic.

We have found three kinds of approaches in the literature. Some of them intend to consider the concurrent tasks altogether to get insight into their possible interferences. Unfortunately, these techniques would probably not be feasible for a real-size system. The second category of approaches includes those that exploit the knowledge of the whole set of concurrent tasks to statically partition accesses to storage and bandwidth resources. This seems to be sound for storage resources, even if it requires a preliminary analysis of conflicts that may be costly in time. But fine-grained static-scheduling schemes for bandwidth resources do not

fit static WCET analysis techniques. For these reasons, approaches belonging to the third category, that aim at making the WCET of one task computable independently of the nature of concurrent tasks, seem to be the most relevant today. However, existing schemes probably do not scale well and will have to be improved to allow wider parallelism.

Research on WCET analysis and WCET-aware design of parallel architectures is still in early stages. We expect these topics to receive more and more attention in the next years. We believe that future critical system designs will favor task isolation at various levels to keep the problem of determining the WCETs of tasks tractable even on large-scale architectures. Task isolation may be enforced using hardware arbitration schemes in a hierarchical architecture where each resource is shared by only a limited number of nodes. In addition, the software should be designed in such a way that conflicts can only occur in well-delimited parts of the task codes. Such a behavior can be achieved considering appropriate resource access models, where a task can access a shared resource only in dedicated phases, as proposed in [36]. Provided the hardware and software conjunctly limit the conflicts between tasks, the techniques that have been proposed to analyse the WCETs considering the possible task interaction may be usable and useful to take into account the remaining possible conflicts.

References

1 J. Barre, C. Rochange, P. Sainrat. *An Architecture for the Simultaneous Execution of Hard Real-Time Threads*. Int'l Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS), 2008.

2 R. Bourgade, C. Rochange, M. de Michiel, P. Sainrat. *MBBA: a Multi-Bandwidth Bus Arbiter for hard real-time*. 5th Int'l Conf. on Embedded and Multimedia Computing (EMC), 2010.

3 C. Burguiere, J. Reineke, S. Altmeyer. *Cache Related Preemption Delay for Set-Associative Caches*. 9th Int'l Workshop on WCET Analysis, 2009.

4 J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, S. Baruah. *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*. In Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/ CRC Press, 2004.

5 F. Cazorla, A. Ramirez, M. Valero, P. Knijnenburg, R. Sakellariou, E. Fernandez. *QoS for High-Performance SMT Processors in Embedded Systems*. IEEE Micro, 24(4), 2004.

6 P. Cousot, R. Cousot. *Static determination of dynamic properties of programs*. 2nd International Symposium on Programming, 1976.

7 P. Crowley, J.-L. Baer. *Worst-Case Execution Time estimation for Hardware-assisted Multithreaded Processors*. 2nd Workshop on Network Processors, 2003.

8 R.I. Davis, A. Burns. *A Survey of Hard Real-Time Scheduling for Multiprocessor Systems*. Accepted for publication in ACM Computing Surveys.

9 S.A. Edwards, S. Kim, E.A. Lee, H.D. Patel, M. Schoeberl. *Reconciling Repeatable Timing with Pipelining and Memory Hierarchy?* Workshop on Reconciling Performance with Predictability (RePP), 2009.

10 A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, B. Lisper. *Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis*. 7th Int'l Workshop on WCET Analysis, 2007.

11 C. Ferdinand, R. Wilhelm. *Fast and efficient cache behavior prediction for real-time systems*. Journal on Real-Time Systems, 17(2/3), Springer, 1999.

12 D. Hardy, T. Piquet, I. Puaut. *Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches*. IEEE Real-Time Systems Symp. (RTSS), 2009.

13 D. Hardy, I. Puaut. *WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches*. IEEE Real-Time Systems Symposium (RTSS), 2008.

14 A. Hansson, K. Goossens, M. Bekooij, J. Huisken. *Compsoc: A template for composable and predictable multi-processor system on chips*. ACM Transactions on Design Autom. Electron. Syst., 14(1), 2009.

15 N. Holsti. *Analysing Switch-Case Tables by Partial Evaluation*. 7th Int'l Workshop on WCET Analysis, 2007

16 B. Lesage, T. Hardy, I. Puaut. *Shared Data Cache Conflicts Reduction for WCET Computation in Multi-Core Architectures*. Int'l Conf. on real-Time Networks and Systems, 2010.

17 Y.-T. S. Li, S. Malik. *Performance Analysis of Embedded Software using Implicit Path Enumeration*. Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.

18 X. Li, A. Roychoudhury, T. Mitra. *Modeling out-of-order processors for WCET analysis*. Journal of Real-Time Systems, 34(3), Springer, 2006.

19 B. Lickly, I. Liu, S. Kim, H.D. Patel, S.A. Edwards, E.A. Lee. *Predictable programming on a precision timed architecture*. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2008.

20 T. Lundqvist, P. Stenström. *Timing Anomalies in Dynamically Scheduled Microprocessors*. IEEE Real-Time Systems Symposium (RTSS), 1999.

21 M. de Michiel, A. Bonenfant, H. Cassé, P. Sainrat. *Static loop bound analysis of C programs based on flow analysis and abstract interpretation*. IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2008.

22 J. Mische, I. Guliashvili, S. Uhrig, T. Ungerer. *How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT*. 23rd Int'l Conf. on Architecture of Computing Systems (ARCS), 2010.

23 M. Paolieri, E. Quinones, F. Cazorla, G. Bernat, M. Valero. *Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems*.36th Int'l Symp. on Computer Architecture (ISCA), 2009.

24 M. Paolieri, E. Quinones, F. Cazorla, M. Valero. *An Analyzable Memory Controller for Hard Real-Time CMPs*. IEEE Embedded Systems Letters, 1(4), 2009.

25 S. Plazar, P. Lokuciejewski, P. Marwedel. *WCET-aware Software based Cache Partitioning for Multi-task Real-time Systems*. 9th Int'l Workshop on WCET Analysis, 2009.

26 I. Puaut. *WCET-centric software-controlled instruction caches for hard real-time systems*. 6th Int'l Workshop on WCET Analysis, 2006.

27 I. Puaut, D. Decotigny. *Low-complexity Algorithms for Statc Cache Locking in Multitasking Hard Real-Time Systems*. IEEE Real-Time Systems Symposium (RTSS), 2002.

28 P. Puschner, A. Burns. *Writing temporally predictable code*. 7th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems, 2002.

29 P. Puschner, M. Schoeberl. *On Composable System Timing, Task Timing, and WCET Analysis*. 8th Int'l Workshop on WCET Analysis, 2008.

30 R. Reddy, P. Petrov. *Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems*. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2007.

31 J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, B. Becker. *Definition and Classification of Timing Anomalies*. 6th Int'l Workshop on WCET Analysis, 2006.

32 C. Rochange, P. Sainrat. *A Context-Parameterized Model for Static Analysis of Execution Times*. Transactions on High-Performance Embedded Architectures and Compilers, 2(3), Springer, 2007.

33 J. Rosén, A. Andrei, P. Eles, Z. Peng. *Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip*. 28th IEEE Int'l Real-Time Systems Symposium, 2007.

34 J. Schneider, C. Ferdinand. *Pipeline behavior prediction for superscalar processors by abstract interpretation*. SIGPLAN Notices, 34(7), ACM, 1999.

35 M. Schoeberl, P. Puschner. *Is Chip-Multiprocessing the End of Real-Time Scheduling?*. 9th Int'l Workshop on WCET Analysis, 2009.

36 A. Schranzhofer, R. Pellizzoni, Jian-Jia Chen, L. Thiele, M. Caccamo. *Worst-case response time analysis of resource access models in multi-core systems*. Design Automation Conference (DAC), 2010.

37 V. Suhendra, T. Mitra. *Exploring Locking and Partitioning for Predictable Shared Caches on Multi-cores*. 45th Conf. on Design Automation (DAC), 2008.

38 D. Tullsen, S. Eggers, H. Levy. *Simultaneous Multithreading: Maximizing On-Chip Parallelism*. 22nd Int'l Symposium on Computer Architecture (ISCA), 1995.

39 T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, *et al.*. *MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability*. IEEE Micro, 30(5), 2010.

40 J. Yan, W. Zhang. *WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches*. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), 2008.

41 Yan Li, V. Suhendra, Yun Liang, T. Mitra, A. Roychoudhury. *Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores*. IEEE Real-Time Systems Symp. (RTSS), 2009.

Towards the Implementation and Evaluation of Semi-Partitioned Multi-Core Scheduling

[Work in Progress]

Yi Zhang[1], Nan Guan[2], and Wang Yi[2]

1 **Northeastern University, China**
2 **Uppsala University, Sweden**

Abstract

Recent theoretical studies have shown that partitioning-based scheduling has better real-time performance than other scheduling paradigms like global scheduling on multi-cores. Especially, a class of partitioning-based scheduling algorithms (called semi-partitioned scheduling), which allow to split a small number of tasks among different cores, offer very high resource utilization, and appear to be a promising solution for scheduling real-time systems on multi-cores. The major concern about the semi-partitioned scheduling is that due to the task splitting, some tasks will migrate from one core to another at run time, and might incur higher context switch overhead than partitioned scheduling. So one would suspect whether the extra overhead caused by task splitting would counteract the theoretical performance gain of semi-partitioned scheduling.

In this work, we implement a semi-partitioned scheduler in the Linux operating system, and run experiments on a Intel Core-i7 4-cores machine to measure the real overhead in both partitioned scheduling and semi-partitioned scheduling. Then we integrate the obtained overhead into the state-of-the-art partitioned scheduling and semi-partitioned scheduling algorithms, and conduct empirical comparison of their real-time performance. Our results show that the extra overhead caused by task splitting in semi-partitioned scheduling is very low, and its effect on the system schedulability is very small. Semi-partitioned scheduling indeed outperforms partitioned scheduling in realistic systems.

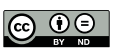
1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: Real-time and embedded systems

Keywords and phrases real-time operating system, multi-core, semi-partitioned scheduling

Digital Object Identifier 10.4230/OASIcs.PPES.2011.42

1 Introduction

It has been widely believed that future real-time systems will be deployed on multi-core processors, to satisfy the dramatically increasing high-performance and low-power requirements. There are two basic approaches for scheduling real-time tasks on multiprocessor/multi-core platforms [3]: In the *global* approach, each task can execute on any available processor at run time. In the *partitioned* approach, each tasks is assigned to a processor beforehand and during the run time each task can only execute on this particular processor. Recent studies showed that the partitioned approach is superior in scheduling hard real-time systems, for both theoretical and practical reasons. However, partitioned scheduling still suffers from resource waste similar to the bin-packing problem: a task would fail to be partitioned to any of the processors when the total available capacity of the whole system is still large. When



Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 42–46

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

the individual task utilization is high, this waste could be significant. In the worst-case only half of the system resource can be utilized in partitioned scheduling.

To overcome this problem, recently researchers proposed *semi-partitioned scheduling* [1, 2, 4, 5, 6, 7], in which most tasks are statically assigned to a corresponding fixed processor as in partitioned scheduling, while a few number of tasks are split into several subtasks, which are assigned to different processors. Theoretical studies have shown that semi-partitioned scheduling can significantly improve the resource utilization over partitioned scheduling, and appears to a promising solution for scheduling real-time systems on multi-cores.

While there have been quite a few works on implementing global and partitioned scheduling algorithms in existing operating systems and studying their characterizations like run-time overheads, the study of semi-partitioned scheduling algorithms is mainly on the theoretical aspect. The semi-partitioned scheduling has not been accepted as a mainstream design choice due to the lack of evidences on its practicability. Particularly, in semi-partitioned scheduling, some tasks will migrate from one core to another at run time, and might incur higher context switch overhead than partitioned scheduling. So one would suspect whether the extra overhead caused by task splitting would counteract the theoretical performance gain of semi-partitioned scheduling.

In this work, we consider the implementation and characterization of semi-partitioned scheduling in realistic systems. We implement a semi-partitioned scheduler in Linux 2.6.32. Then we measure its realistic run-time overhead on an Intel Core-i7 4-cores machine. Finally we integrate the measured overhead into empirical comparison of the state-of-the-art partitioned scheduling and semi-partitioned scheduling algorithms. Our experiments show that semi-partitioned scheduling indeed outperforms partitioned scheduling in the presence of realistic run-time overheads.

2 Implementation of Semi-Partitioned Scheduler

Several semi-partitioned algorithms have been proposed [4]. In this work we adopt a recent developed algorithm FP-TS [4], which is based on Rate-Monotonic Scheduling. FP-TS has both high worst-case utilization guarantees (can achieve hight utilization bounds) and good average-case real-time performance (exhibits high acceptance ratio in empirical evaluations). A detailed description of FP-TS can be found in [4]. Our semi-partitioned scheduler implementation can be easily extended to support a wide range of semi-partitioned algorithms based on both fixed-priority and EDF scheduling.

Now we introduce our semi-partitioned scheduler implementation in Linux 2.6.32. The basic framework of our semi-partitioned scheduler is as follows: Each core has its own Ready queue, which records the tasks have been released but not finished on this core. When a task is released, it will be inserted into the ready queue, and trigger the scheduler. The scheduler decides the task to be executed according to the priority order. The timing parameters of each task are stored in the date structure *task_struct* when the task is created.

There are two types of tasks in the system: (1) *normal tasks*, which execute on a fixed core, and (2) *split tasks*, which will migrate among different cores. The main challenge of the semi-partitioned scheduling is to support splitting tasks to correctly execute on different cores, and to migrate from one core to another core with the timing constraint (obtained from the partitioning algorithm) with as small as possible run-time overhead.

In our implementation, each core maintains its own sleep queue, which records tasks on this core that are currently not active, and its own ready queue, which records tasks on this core that are currently active. The ready queue is implemented by a binomial heap and the

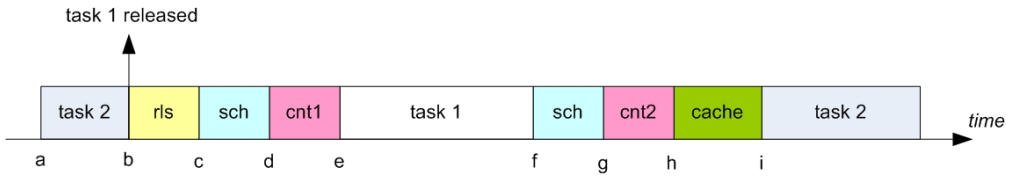


Figure 1 An example to illustrate the run-time overhead.

sleep queue is implemented by a red-black tree. For a split task, we need to control when a subtask on one core will migrate to another. This is done by recording the time budget in the split task's *task_struct* data structure. The main difference between normal tasks and split tasks is in the scheduling action after their budgets on this core are run out. If it is a normal task, the scheduler will put this task to the sleep queue of this core. If it is a split task, the scheduler will: (1) if it is a body subtask, the scheduler will insert the next subtask into the ready queue of the migration destination core, and trigger the scheduling on the destination core; (2) if it is a tail subtask, the scheduler will put this task back to the sleep queue of the core hosting the first subtask of this split task.

3 Overhead Measurement

We use the example in Figure 1 to illustrate the overhead that may happen at runtime. We assume at time a a lower-priority task τ_2 is executing, and at time b, a higher-priority task τ_1 is released. The time between b and e is the overhead due to the release of τ_1 and context switch from τ_2 to τ_1. Task τ_1 finishes its execution at time f, and the time between f and i is the overhead due to the context switch from τ_1 and τ_2. From time i, τ_2 continue to execute the unfinished work. Now we introduce different parts of the overhead one by one.

- ***rls***: This is the overhead due to the task release: When a task is released, the function *release*() is invoked to insert this task into the ready queue. ***rls*** includes the delay from requesting the access to getting access to the ready queue, and the time of doing the insert operation on the ready queue.
- ***sch***: This is the overhead due to the scheduling actions, which is in the function *sch*(). It may happen in two cases: (1) Task release. In this case, *sch*() will select the highest-priority task from the ready queue. If there happens a preemption, *sch*() will put the current running back to the ready queue. (2) Task finish. In this case, *sch*() will select the highest-priority task from the ready queue.
- ***cnt1***: This is the overhead due to the context switch from the preempted task to the preempting task, which is in the function *cnt_swth*(). It will store the preempted task's context and load the preempting tasks's context.
- ***cnt2***: This overhead is also in the function *cnt_swth*(). It may happen in three cases: (1) The current task is a normal task, and has finished its work. In this case, *cnt_swth*() will load the context of the task to run (the highest-priority task selected by *sch*()), then insert the finished task into the sleep queue. (2) The current task is a split task, and it has run out of its budget on this core and will migrate to another. In this case, *cnt_swth*() will reload the context of the task to run next, then insert this task to the ready queue of the destination core. (3) The current task is a split task, and it has finished its execution. In this case, *cnt_swth*() will reload the context of the task to run next, then insert this task into the sleep queue of core which hosts the first subtask of this split task.

Operation	local ($N = 4$)	remote ($N = 4$)	local ($N = 64$)	remote ($N = 64$)
sleep queue – add	2.5	2.9	4.3	4.4
sleep queue – delete	**3.3**	N/A	**5.8**	N/A
ready queue – add	1.5	**3.3**	4.4	**4.6**
ready queue – delete	2.7	N/A	4.6	N/A

Table 1 The measured queue operation durations, all in μs

.

- ***cache***: The preempted task's working space would be (partially) replaced out from the cache, and when it resumes execution, it needs to reload its working space.

The table shows the maximal measured duration of a single ready queue operation and sleep queue operation. We set θ and δ to be the worst-case value among them: when $N = 4$, $\delta = 3.3\mu s$ and $\theta = 3.3\mu s$; when $N = 64$, $\delta = 4.6\mu s$ and $\theta = 5.8\mu s$ (N is the maximal number of tasks in the queue, i.e., the number of tasks on this core). Apart from the delay due to the access to the ready and sleep queues, we also measure the pure execution time of the functions $relase()$, $sch()$ and $cnt\_swth()$, they are $3\mu s$, $5\mu s$ and $1.5\mu s$ respectively.

The last overhead we measured is the cache-related overhead. This overhead is highly dependent on the application memory characters. An important issue is the difference between local context switches and task migrations between cores. Our measurement shows that in general the cache-related overhead due to task migrations and local context switches is in the same order of magnitude. This is due to the shared lower-hierarchy caches (L3 cache in our case): in both local context switches and task migrations, most of the working space of the preempted/to-migrate task will be replaced out from the private cache (L1 and L2 cache in our case), and stay in the shared lower-hierarchy caches. Of course, if an application has generally very small working space (much smaller than the size of private cache, which is rather rare in realistic applications), the cache-related delay of local context switches would be significantly smaller than task migrations, since there is a better chance for the working space of the preempted task to stay in the private cache, until it resumes execution.

4 Results and Conclusion

We conduct comparison of the performance in terms of acceptance ratio of FP-TS and two widely used fixed-priority partitioned scheduling algorithm FFD (first-fit decreasing size partitioning) and WFD (worst-fit decreasing size partitioning), with randomly generated task sets, taking into account the measured overheads shown in last section. Our experiments show that semi-partitioned scheduling indeed outperforms partitioned scheduling in the presence of realistic run-time overheads.

References

1 B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems multiprocessors. In *RTSS*, 2008.

2 B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA*, 2006.

3 J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. 2004.

4 N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In *RTAS*, 2010.

5 S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *EMSOFT*, 2008.
6 S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *RTAS*, 2009.
7 S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *ECRTS*, 2009.

An Automated Flow to Map Throughput Constrained Applications to a MPSoC*

Roel Jordans[1], Firew Siyoum[1], Sander Stuijk[1], Akash Kumar[1,2], and Henk Corporaal[1]

1 **Eindhoven University of Technology, The Netherlands**
2 **National University of Singapore, Singapore**

Abstract

This paper describes a design flow to map throughput constrained applications on a Multi-processor System-on-Chip (MPSoC). It integrates several state-of-the-art mapping and synthesis tools into an automated tool flow. This flow takes as input a throughput constrained application, modeled with a synchronous dataflow graph, a C-based implementation for each actor in the graph, and a template based architecture description. Using these inputs, the tool flow generates an MPSoC platform tailored to the application requirements and it subsequently maps the application to this platform. The output of the flow is an FPGA programmable bit file. An easily extensible template based architecture is presented, this architecture allows fast and flexible generation of a predictable platform that can be synthesized using the presented tool flow. The effectiveness of the tool flow is demonstrated by mapping an MJPEG-decoder onto our MPSoC platform. This case study shows that our flow is able to provide a tight, conservative bound on the worst-case throughput of the FPGA implementation. The presented tool flow is freely available at `http://www.es.ele.tue.nl/mamps`.

1998 ACM Subject Classification C.3 [Special-purpose and application-based systems]: real-time embedded systems

Keywords and phrases design flow automation, multi-processor system-on-chip, throughput constrained, synchronous data-flow graphs

Digital Object Identifier 10.4230/OASIcs.PPES.2011.47

1 Introduction

New applications for embedded systems demand complex multiprocessor designs to meet real-time deadlines while achieving other critical design constraints like low energy consumption and low area usage. Multiprocessor Systems-on-Chip (MPSoCs) have been proposed as a promising solution for such problems but the design space exploration of such systems typically involves many parameters. Higher abstraction levels, possibly combined with early and accurate performance predictions, of the designed system are therefore required to make good design choices. Several tool-flows [6, 10, 13] have been proposed to solve this problem, but these solutions still require manual design steps which are time consuming and error-prone. Combining existing tools into a common design flow has proven non-trivial [12] without careful planning and coordination of the tool development.

In this paper, we present a design flow (see Figure 1) which bundles the strengths of both the SDF^3 [14] tool set and the MAMPS [8] platform. The SDF^3 tool set supports analyzing

* This work was partly funded by the PROGRESS program of the Dutch Technology Foundation STW through the PreMaDoNa project EES. 6390.

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 47–58

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

and mapping synchronous data-flow (SDF) graphs [9]. SDF^3 uses a graph representation of the application and a set of models of the hardware platform to calculate the worst-case throughput of the application for a given mapping of tasks on the platform. MAMPS provides a tool to generate MPSoC projects for a Xilinx FPGA platform including software and hardware synthesis based on a SDF description of one or more applications and a task mapping. MAMPS has been almost completely rewritten as part of this work, new communication options have been added and the generated hardware and software have been modeled into SDF^3. This ensures that the MAMPS implementation of any mapping produced by SDF^3 can be guaranteed to meet or exceed the throughput guarantee provided by SDF^3 and thus produce a predictable system.

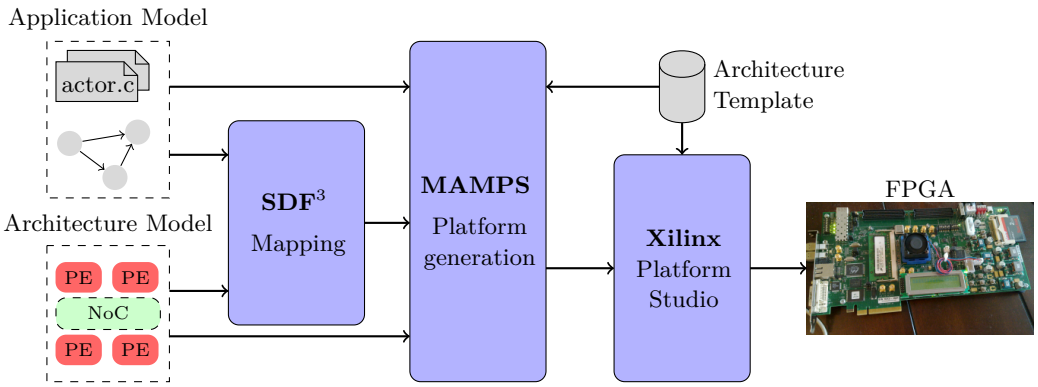


Figure 1 Design flow overview.

The remainder of this paper is organized as follows. Section 2 reviews the related work for automated MPSoC generation and performance prediction. Section 3 provides an overview of application modeling using SDF graphs. Section 4 gives an overview of the architecture of the MAMPS platform. The design flow is presented in Section 5 and the implementation changes to both SDF^3 and MAMPS are explained in this section. Section 6 presents an experiment used to validate the design flow and analyzes the design effort and design overhead of the flow. Section 7 concludes the paper and gives a direction for future work.

2 Related work

The problem of mapping an application to an architecture has been widely studied in literature. One of the recent works most related to our research is CA-MPSoC [13]. CA-MPSoC extends the MAMPS platform with a hardware communication assist (CA) which is responsible for the communication between the processing elements of the platform. The paper presents a SDF model for this CA controller and uses this model for performance prediction. However, the presented model has been simplified and lacks modeling of the communication channel. This paper improves the model by *a*) including the fragmentation of communicated tokens into words that can be sent over a network, and *b*) including a model for the communication channel on the network itself. The flow presented in [13] introduces options for deciding on a mapping of the application onto the generated platform but the method requires the user to manually translate the output format of the mapping tool into the interchange format of the platform generation tool. The flow presented in this paper automates this step by introducing a common input format for both the mapping and platform generation tools, circumventing possible user introduced errors during the translation step.

ESPAM [10, 11] presents a similar design flow as the flow presented in this paper. The ESPAM flow uses Kahn Process Networks (KPNs) to model the application. In our approach, we use SDF graphs in stead. SDF graphs are a subset of KPN graphs and therefore have a limited expressiveness when compared to KPN graphs. The disadvantage of using pure KPN for application modelling however is the limited possibilities for analyzing pure KPN graphs. It is, for example, impossible to analyze buffer requirements in a generic way when using KPN graphs but this analysis is possible for SDF graphs [2]. Another disadvantage of KPN over SDF is that KPN requires run-time buffer management and scheduling which make performance prediction difficult while SDF graphs can be completely analyzed at design time [14]. Our approach produces a predictable, throughput constrained solution whereas ESPAM is limited to an estimation of the performance. The PeaCE approach presented in [6] provides another method for hardware and software co-design. PeaCE uses two different extended versions of the SDF model and three different types of tasks for representing different parts of the application, requiring a relatively complex operating system. Our approach uses a pure SDF representation of the application and implements only a single task type resulting in a minimal implementation overhead. This comes however at the cost of a reduced expressiveness and therefore potentially an over dimensioning of our platform. The experimental results show however that this effect is limited.

3 Application Modelling

Figure 2 shows an example of an SDF graph. There are three actors in this graph. As in a typical data flow graph, a directed edge represents the dependency between actors. Actors consume input data from their input edges and/or produce output data on their output edges; such information is referred to as *tokens*. Tokens are shown in an SDF graph as dots on the edges, a number is added to these dots to show that multiple tokens are available. The number of tokens consumed by an actor is constant and can be read from the SDF graph next to the incoming vertex. An actor is called *ready* when it has sufficient input tokens on all its input edges. Actor execution is called *firing*, an actor can only fire when it is ready. An actor also produces a constant amount of tokens per firing denoted next to the outgoing end of each edge. SDF actors are stateless (i.e. no internal actor state is preserved between actor firings) so any actor state need to be modelled explicitly. Actor A in Figure 2 is an example of an actor which keeps state, implemented as the static variable in Listing 1, this state variable is modeled explicitly in Figure 2 by the self-edge of actor A.

Listing 1 Implementation of actor A

```
static int local_variable_A;

void actor_A_init(typeAtoB *, typeAtoC *) {
   local_variable_A = 0;
}

void actor_A (typeAtoB *toB, typeAtoC *toC) {
   // calculate something
   // and write the output tokens
   toB[0] = calculate_valueB1();
   toB[1] = calculate_valueB2();
   *toC = calculate_valueC(local_variable_A);
}
```

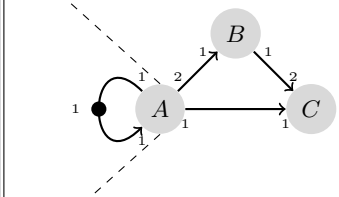


Figure 2 Example of an SDF graph together with the implementation of one of the actors.

An application is described using a graph. Edges may contain *initial tokens* as is shown on the self-edge of actor A in Figure 2. In the above example, only A can fire in the initial

state, since the required number of tokens are present on all of its incoming edges. Once A has finished firing it will produce 2 tokens on its edge to B, 1 token on its edge to C and 1 token on its self-edge. B can then fire as it has enough tokens on its incoming edge to execute twice, each time producing 1 token on its edge to C.

Implementing an application using its SDF graph requires an implementation for each actor. Actor implementations consist of one actor implementation function which takes up to one parameter per edge connected to the actor. Not every edge needs to be explicitly implemented as a parameter to the actor implementation function. The self-edge of actor A is an example of an edge which is not explicitly implemented. Therefore we make a distinction between explicitly and implicitly implemented edges. Explicitly implemented edges implement connections between two actors which are transferring data. Implicitly implemented edges include, but are not limited to, the self-edges as shown above, but can also be used to model restrictions like limited buffer sizes on the edges connecting multiple actors as well as modeling a specific firing order as imposed by static order scheduling [14]. Only explicit edges are implemented as parameters of the actor implementation function. Listing 1 shows an example implementation of actor A. Two functions are created in this listing, an initialization function and the actor implementation. The actor implementation function `actor_A()` has two parameters, one for the edge to B and one for the edge to C, note that there are no parameters supplied for the implicit self-edge of A. Output tokens are written to the buffers provided as parameters. The initialization function, `actor_A_init`, is responsible for producing the initial tokens that are expected on the output edges of actor A, in this case the self-edge of A. The initialization function has the same signature as the main actor implementation but no space is reserved for edges that do not produce initial tokens and no input tokens are provided.

The application graph and the relation between the graph elements and their respective implementations are joined into the *application model*. The application model also specifies a set of metrics of the actor implementations. These metrics include the Worst-Case Execution Time (WCET), required memory sizes, and the size of communicated tokens. Memory size requirements are specified separately for both instruction and data memories in order to facilitate processing elements that use a Harvard architecture. The memory size requirement is used in the tool flow to automatically determine the memory requirements for each processing element. The WCET metrics and token sizes are used by the SDF^3 tools to calculate a lower bound on the throughput of the application. A good WCET estimate of each actor implementation is therefore important for the performance of the presented tool flow. Many different approaches exist for determining the WCET of (a part of) a program, either from the source code or some intermediate form. WCET tool challenges [5, 7] present insightful information about existing WCET analysis tools and techniques and [16] gives an in-depth analysis of the available methods as well as a survey of existing tools for WCET analysis. Any of these tools can be used to provide the WCET of actors for the presented design flow. It is possible that different (optimal) implementations of the same actor exist for the different types of processing element or tile configuration in the platform template. The application model can specify multiple implementations for each actor. Each implementation specification defines the relation between the function arguments of the implementation and the edges of the graph, the WCET and memory requirements of that specific implementation, and the type of processing element this implementation can be mapped to. This allows the tool flow to map the actors on a heterogeneous platform where actor implementations for different processing elements are likely to have different metrics.

4 Architecture Modelling

The second input of the design flow is the *architecture model* (see Figure 1). This model describes the various components available in the hardware platform and how these components are connected. The MAMPS platform allows two types of components in the architecture; tiles and interconnect. Tiles form the processing elements of the architecture and the interconnect is limited to connecting tiles together. A standardized network interface (NI) has been defined for connecting tiles to the interconnect. All tile and interconnect variants use this same network interface which makes it easy to compose a platform by using elements from an architecture template. Figure 3 shows an example of the MAMPS platform architecture. This example shows four different variations of a tile connected together through an interconnect. Tiles 1 and 2 in the example show simple tile architectures using a processing element (PE) which is connected to the network interface (NI), a local memory and some optional peripherals (i.e. IO, timers, etc.). Tile 3 shows a similar tile which has been extended with a communication assist (CA) which handles the memory management and serialization, sending, and receiving of tokens. The last tile, Tile 4, shows another option where a hardware implementation of an actor (IP) is connected directly to the interconnect using only a network interface.

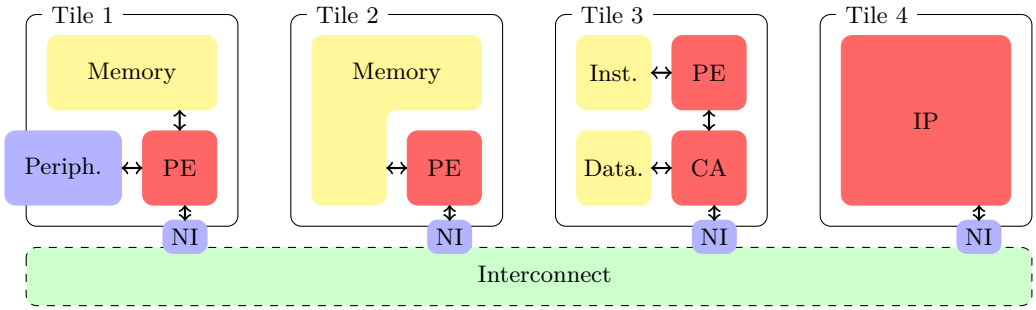


Figure 3 MAMPS platform architecture example showing different variations of a tile.

Running realistic applications on a system requires that one or more actors have access to peripherals. Predictability of the MAMPS platform is guaranteed by avoiding the sharing of peripherals over tiles. Another option for maintaining predictability while using shared peripherals is to use a predictable arbiter. [1] presents such an arbiter for SDRAM memories. The technique presented in [1] can be extended to include different types of resources and is easy to implement.

4.1 Network interface

A clear definition of the network interface is critical for the functioning of the template based architecture generation. The MAMPS platform defines the Xilinx Fast Simplex Link interface as network interface. This limits the network interface to communicating 32-bit words but also makes sure there is a trivial point-to-point solution for the interconnect by using Xilinx Fast Simplex Links (FSL) [15]. In order to translate arbitrarily sized tokens into one or more 32-bit words and back again requires serialization and de-serialization. These operations can either be performed by the processing element of the tile (i.e. the PE block in Tile 1 of Figure 3), or by the addition of some dedicated communication hardware (i.e. the CA block of Tile 3 in Figure 3).

The advantage of using the processing element for the serialization and de-serialization

of tokens is the simplicity of the generated hardware. This simple hardware comes at the cost of extra processing time used on the processing element which can not be spent on running actor code. Using dedicated communication hardware like the CA described in [13] increases hardware complexity but also relieves the processing element from the serialization and de-serialization of tokens which improves the actor response-time.

4.2 Communication model

The communication introduced in MAMPS has been modeled in an SDF graph. This graph is used in SDF^3 to predict the behaviour of edges mapped to the interconnect. Figure 4 shows the parameterized model of communication via the interconnect. Three boxes divide this model into the parts representing the various phases in the communication of a token. The dashed edge in this graph shows the original connection in the SDF graph.

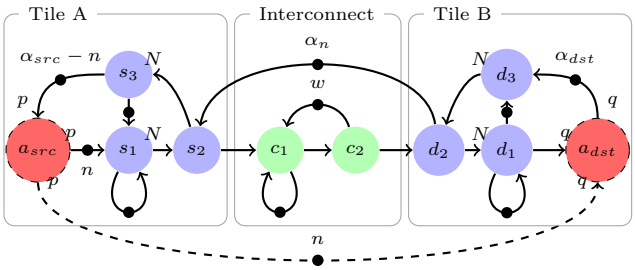


Figure 4 Parameterized model for communication over the interconnect. Missing port rates and token counts are to be interpreted as 1.

The central box models the interconnect behaviour, the model allows pipelined sending of words over the interconnect where the number of initial tokens w is equal to the maximum number of words in simultaneous transmission. The connections on the interconnect are also capable of buffering a number of α_n words in transmission. Actors c_1 and c_2 form a latency-rate model for the communication. Actors s_1, s_2 and s_3 model the serialization of the token into N 32-bit words by the network interface. The execution time of s_1 is dependant on the design of the serialization code while the execution times for s_2 and s_3 are set to 0 because these actors are only required for the modeling of the serialization of the tokens. Actors d_1, d_2 and d_3 model the de-serialization of the transmitted words into tokens at the receiving end and are assigned values in the same way as the serialization actors. Finally, α_{src} and α_{dst} model the available buffer space on the sending and receiving ends of the connection. The model in Figure 4 can be used for modeling communication over many different forms of interconnect by changing w, α_n, and the execution times of s_1, c_2, and d_1 to appropriate values.

5 Design flow

The design flow, as depicted in Figure 1 can be divided in three steps. The application should first be mapped onto the architecture. This mapping can then be combined with the original application and architecture specifications into a FPGA design which can, as a third step, be synthesized into a working system using out of the box FPGA development software. The goal of this flow is to produce a working implementation of the application on a given platform, capable of achieving the throughput as required for the application. The *throughput* of an application is defined in [3] as the long term average number of graph

iterations per time unit. The long term average is used to avoid initialization effects from influencing the throughput. The design flow defines the system clock of the platform as its base time unit. This section provides a more in-depth description of the SDF^3 tool set, the MAMPS platform generation, and the currently available architecture components.

5.1 SDF^3

The SDF^3 tool set consists of several tools that allow automatic mapping of an application described as a SDF graph to a given platform. SDF^3 also verifies if such a mapping is deadlock free, calculates buffer distributions, and predicts which throughput can be guaranteed for this mapping. SDF^3 uses generic cost functions to steer the binding of the application to the architecture based on; processing, memory usage, communication, and latency. Buffer distributions, task mapping and static-order schedules are determined and gathered in the mapping output of SDF^3. The virtual platform of the SDF^3 tool set was modified to match the architecture and model of the MAMPS platform. The algorithms used during mapping have not been changed from those presented in [14].

5.2 MAMPS

The MAMPS tool set was completely rewritten as part of this research but the architecture and ideas remain the same. The platform is now generated by combining the information from the application and architecture models with the mapping output from SDF^3. Information from the architecture model and mapping are used to generate the hardware platform. Template components are instantiated and connected as required by the application. Memory sizes are calculated for each tile based on the mapped buffers, actors and the size of the scheduling and communication layer. The interconnect components are instantiated to match the specified communication architecture. Connections are routed and the VHDL code and peripheral driver for the interconnect are also generated when required. The software platform is generated next. This includes generating wrapper code for each actor, translating the static-order schedule provided by SDF^3 into C code, and generating initialization code for the communication. The generated code is combined with a template project which already includes an implementation of the scheduling and communication libraries. The XPS TCL script interface is then used to complete the project and to add the required hard and software targets for the implementation. Using the script interface ensures compatibility over many different versions of XPS and greatly simplifies the generated code.

5.3 Currently available in the architecture template

Not all blocks shown in Figure 3 are currently available in the tool flow. The MAMPS platform currently offers two forms of interconnect and two different tiles. The currently available architecture components are all targeted for the Xilinx Virtex6 FPGA using the Xilinx ML605 evaluation board. The subsections below discuss the available options.

5.3.1 Interconnect

Either point-to-point connections using Xilinx Fast Simplex Links (FSL) [15] or a Spatial Devision Multiplex (SDM) NoC based on [17] can be used for connecting the tiles. Both interconnects comply to the network interface definition but the NoC interconnect provides more flexibility at the cost of a larger implementation and a higher latency while the FSL interconnect simply uses the FSL implementation provided by Xilinx.

The NoC consists of one router per tile in the design. Each router connects through a set of wires to its neighbours. Each router can also be connected to the network interface of a single tile. The routers are arranged in a 2-dimensional mesh network. The dimensions of this network are based on the number of tiles required in the design and the network is kept as close to square as possible to reduce the maximum distance between two tiles since this distance relates directly to the latency of the network connections. The NoC allows the user to program connections on a point-to-point basis, each connection can be assigned a certain bandwidth through the number of wires assigned to that connection but wires can only be assigned to a single connection at a given time allowing an efficient usage of network resources. The original NoC presented in [17] already complied with the network interface requirements for the MAMPS platform but missed flow-control for connections in the network. Flow-control was added as part of the integration of the NoC in the MAMPS platform. The changes to the NoC required approximately 12% more slices on the FPGA when compared to the original implementation.

5.3.2 Tile template

As shown in Figure 3, a tile consists of a processing element (PE), an optional instruction and/or data memory, and a network interface (NI). MAMPS currently provides only two types of tiles. The first type is the master tile, this tile is similar to Tile 1 in Figure 3. It uses a Xilinx Microblaze soft-core as processing element, includes up to 256kB memory in a Modified Harvard configuration and has direct access to the peripherals on the FPGA board. The FSL ports of the Microblaze and a software library implementing (de-)serialization are used to implement the network interface. The second type of tile is the slave tile, this tile is the same as the master tile but does not have access to the peripherals and therefore is similar to Tile 2 in Figure 3.

6 Case study

The application used in the case study is the MJPEG decoder shown in Figure 5. The VLD actor parses the input file and decompresses the Minimal Coded Unit (MCU) blocks. MCUs consist of up to 10 blocks of frequency values, depending on the sampling settings used when creating the input file. Each block of frequency values is passed through the inverse quantization and zig-zag reordering (IQZZ) and IDCT actors which transform the frequency values into color components. The color conversion (CC) actor translates the color component blocks of one MCU to pixel values and the rasterization (Raster) actor puts the pixel values at the correct location in the output buffer. The subHeader1 and subHeader2 edges in the SDF graph forward information from the file header (i.e. frame size and color composition) to the CC and Raster actors. One graph iteration of the MJPEG decoder decodes a single MCU. This causes the throughput of the application to be defined in MCUs per clock cycle of the generated platform. A method based on [4] combined with execution time measurement was used to determine the WCET of the actors in this case study.

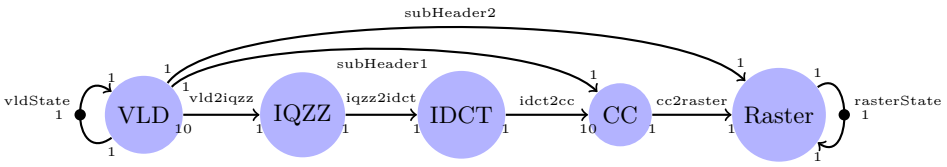
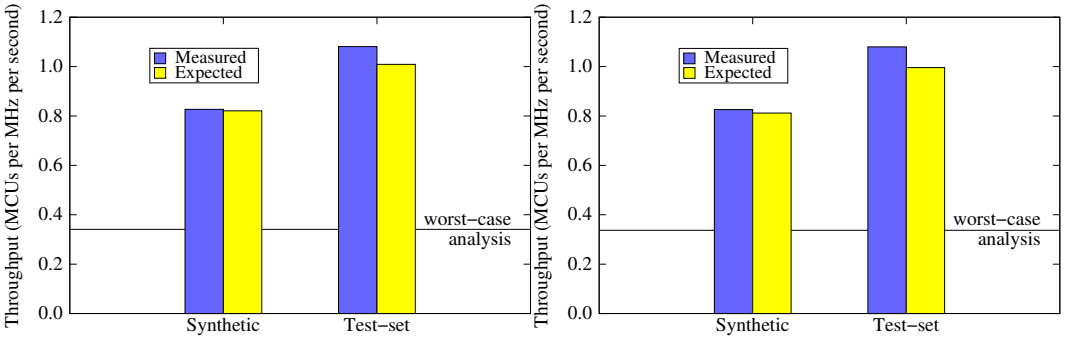


Figure 5 The SDF graph for the MJPEG decoder.

6.1 Throughput analysis

An important aspect of the presented design flow is the early throughput analysis of the designed application. The throughput of the MJPEG decoder was therefore measured on the FPGA implementation and compared to the predicted throughput of SDF^3. Figure 6 shows the worst-case throughput obtained by running the MJPEG decoder on 5 different test sequences and a synthetic sequence containing random data for two different architectures. The worst-case analysis line in both graphs shows the SDF^3 prediction based on the WCET of the actors. The expected values were calculated using SDF^3 by using WCET metrics obtained through execution time measurement of the actor code using the test-data used for the FPGA measurement. The difference between the expected throughput (blue) and measured throughput (yellow) shown in Figure 6 shows the margin of the used models (less than 1% for the synthetic data) when using actors with low variation in the execution time. Throughput at the worst-case analysis line is guaranteed by the flow.



(a) FSL interconnect **(b)** NoC interconnect

Figure 6 Measured and predicted worst-case throughput for a synthetic test-sequence and a set of real-life test-sequences for two different forms of interconnect compared to the worst-case prediction of SDF^3

6.2 Designer effort

Table 1 lists the required designer effort in creating and mapping the MJPEG decoder as this was done by the authors of the paper. This implies a working understanding of the application as well as previous experience in writing applications for the design flow and platform. The top part of the table represents manual labour performed by the designer and the bottom part (marked with A) is automated by the presented design flow. Manually implementing the overall system would cost at least another 2–5 days depending on the complexity of the hardware (i.e. number of tiles) and the number of application mappings tried.

6.3 Overhead

The overhead of the generated system when compared to a manually developed system can be characterized in two categories, modeling and implementation overhead. The primary source of modeling overhead are the fixed output rates of the SDF actors. This can be seen in the MJPEG example at the output rate for the VLD actor which produces up

Table 1 Designer effort, steps marked with A are automated.

Step	Time spent	
Parallelizing the MJPEG code	$<$ 3 days	
Creating the SDF graph	5 minutes	
Gathering required actor metrics	1 day	
Creating application model	1 hour	
Generating architecture model	1 second	A
Mapping the design (SDF^3)	1 minute	A
Generating Xilinx project (MAMPS)	16 seconds	A
Synthesis of the system	17 minutes	A
Total time spent	**$\sim$ 4 days**	

to 10 frequency blocks per MCU depending on the format of the input stream. Another source of modeling overhead can be found in communicating the initialization values on the subHeader1 and subHeader2 channels in the example. A manual implementation of the algorithm could communicate these values separately from the main program flow during an initialization phase, it is not possible to model this using a single SDF graph. However, these initialization tokes are relatively small and use only 1% of the communication. The implementation overhead of SDF is also very small. Scheduling on the MAMPS platform is done through a static order schedule which reduces the scheduler to a lookup table. A manual implementation is likely to implement the same schedule in its main loop which is similar in efficiency. Communication would also be solved in a similar way and therefore does not influence the implementation overhead. The scheduling overhead will be similar for other applications but the modeling overhead and communication overhead will vary depending on the nature of the application.

A short second experiment was performed to study the overhead incurred by the (de-)serialization code in the current tile implementation. In this experiment, the worst-case execution time of the (de-)serialization functions was replaced with the execution time of the communication assist as presented in [13] and the WCET of the (de-)serialization routine was no longer counted towards the execution time of the processing element. This resulted in, according to SDF^3, an increased throughput for our case-study by up to 300% when actors were mapped to the same resources as in the original experiment. This suggests that the use of a CA will greatly improve the usability of the MAMPS platform, but this result could not be verified on hardware because there is currently no support for tiles using a CA.

7 Conclusions

In this paper, we present an automated design flow that is capable of generating an implementation of a given application on a MPSoC and correctly predicting the worst-case performance of the generated implementation. The design flow provides a method for automatically instantiating different architectures using a template based architecture model. This template based architecture is easy to extend and allows the automated selection of the correct implementation when heterogeneous systems are designed. This allows the designers to perform a very fast design space exploration for real-time embedded systems. Together with the publication of this paper the whole flow will be made publicly available to the research community at `http://www.es.ele.tue.nl/mamps`. For future work we would like to offer an improved automated design space exploration and more variation in the

architecture template. Adding a predictable arbiter could enable multiple tiles in accessing peripherals while keeping a predictable system. Finally, we plan to add the communication assist presented in [13].

References

1 Benny Akesson *et al.*: *Predator: a predictable SDRAM memory controller*; in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*; p. 251–256; New York, NY, USA; 2007; ACM.

2 Marc Geilen and Twan Basten: *Requirements on the Execution of Kahn Process Networks*; in *Programming Languages and Systems*; vol. 2618 of *Lecture Notes in Computer Science*; p. 319–334; Springer Berlin / Heidelberg; 2003.

3 Amir Hossein Ghamarian *et al.*: *Throughput Analysis of Synchronous Data Flow Graphs*; in *Proceedings of International Conference on Application of Concurrency to System Design*; p. 25–36; Los Alamitos, CA, USA; 2006; IEEE Computer Society.

4 Stefan Valentin Gheorghita *et al.*: *Automatic scenario detection for improved WCET estimation*; in *Proceedings of the 42nd annual Design Automation Conference*; p. 101–104; New York, NY, USA; 2005; ACM.

5 Jan Gustafsson: *The Worst Case Execution Time Tool Challenge 2006*; in *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*; p. 233 –240; nov. 2006.

6 Soonhoi Ha *et al.*: *Hardware-Software Codesign of Multimedia Embedded Systems: the PeaCE*; in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*; p. 207–214; 2006.

7 Niklas Holsti *et al.*: *WCET 2008 – Report from the Tool Challenge 2008 – 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*; in *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*; Dagstuhl, Germany; 2008; Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

8 Akash Kumar *et al.*: *Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA*; *ACM Transactions on Design Automation of Electronic Systems*; 13(3), p. 1–27; 2008.

9 Edward A. Lee and D.G. Messerschmitt: *Synchronous data flow*; *Proceedings of the IEEE*; 75(9), p. 1235 – 1245; sep. 1987.

10 Hristo Nikolov *et al.*: *Multi-processor system design with ESPAM*; in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*; p. 211–216; oct. 2006.

11 Hristo Nikolov *et al.*: *Systematic and Automated Multiprocessor System Design, Programming, and Implementation*; *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*; 27(3), p. 542–555; mar. 2008.

12 Andy Pimentel *et al.*: *Tool Integration and Interoperability Challenges of a System-Level Design Flow*; in *Embedded Computer Systems: Architectures, Modeling, and Simulation*; vol. 5114 of *Lecture Notes in Computer Science*; p. 167–176; Springer Berlin / Heidelberg; 2008.

13 Ashan Shabbir *et al.*: *CA-MPSoC: An automated design flow for predictable multi-processor architectures for multiple applications*; *Journal of Systems Architecture*; 56(7), p. 265–277; 2010; Special Issue on HW/SW Co-Design: Systems and Networks on Chip.

14 Sander Stuijk: *Predictable Mapping of Streaming Applications on Multiprocessors*; PhD Thesis; Eindhoven University of Technology; 2007.

15 Xilinx website: *Fast Simplex Link overview*; apr. 2010; `http://www.xilinx.com/products/ipcenter/FSL.htm`.

16 Reinhard Wilhelm *et al.*: *The worst-case execution-time problem—overview of methods and survey of tools*; *ACM Transactions on Embedded Computer Systems*; 7(3), p. 1–53; 2008.

17 Zhiyao Joseph Yang *et al.*: *An Area-efficient Dynamically Reconfigurable Spatial Division Multiplexing Network-on-Chip with Static Throughput Guarantee*; in *Proceedings of International Conference on Field Programmable Technology*; p. unknown; Beijing, China; dec. 2010; IEEE; Paper accepted for publication.

Towards Formally Verified Optimizing Compilation in Flight Control Software*

Ricardo Bedin França[1,2], Denis Favre-Felix[1], Xavier Leroy[3], Marc Pantel[2], and Jean Souyris[1]

1 **AIRBUS Operations SAS**
316 Route de Bayonne, Toulouse, France
{ricardo.bedin-franca,denis.favre-felix,jean.souyris}@airbus.com
2 **Institut de Recherche en Informatique de Toulouse**
2 Rue Charles Camichel, Toulouse, France
{ricardo.bedinfranca, marc.pantel}@enseeiht.fr
3 **INRIA Rocquencourt**
Domaine de Voluceau, Le Chesnay, France
xavier.leroy@inria.fr

Abstract

This work presents a preliminary evaluation of the use of the CompCert formally specified and verified optimizing compiler for the development of level A critical flight control software. First, the motivation for choosing CompCert is presented, as well as the requirements and constraints for safety-critical avionics software. The main point is to allow optimized code generation by relying on the formal proof of correctness instead of the current un-optimized generation required to produce assembly code structurally similar to the algorithmic language (and even the initial models) source code. The evaluation of its performance (measured using WCET) is presented and the results are compared to those obtained with the currently used compiler. Finally, the paper discusses verification and certification issues that are raised when one seeks to use CompCert for the development of such critical software.

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors – Compilers, J.2 [Physical Sciences and Engineering] Aerospace

Keywords and phrases Compiler verification, avionics software, WCET, code optimization

Digital Object Identifier 10.4230/OASIcs.PPES.2011.59

1 Introduction

As "Fly-By-Wire" controls have become standard in the aircraft industry, embedded software programs have been extensively used to improve planes' controls while simplifying pilots' tasks. Since these controls play a crucial role in flight safety, flight control software must comply with very stringent regulations. In particular, any flight control software (regardless of manufacturer) must follow the DO-178/ED-12 [1] guidelines for level A critical software: when such software fails, the flight as a whole (aircraft, passengers and crew) is at risk.

The DO-178 advocates precise well-defined development and certification processes for avionics software, with specification, design, coding, integration and verification activities being thoroughly planned, executed, reviewed and documented. It also enforces traceability among development phases and the generation of correct, verifiable software. Verification and

* This work was partially supported by ANR grant Arpège U3CAT.

Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011).
Editors: Philipp Lucas, Lothar Thiele, Benoît Triquet, Theo Ungerer, Reinhard Wilhelm; pp. 59–68

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

tooling aspects are also dealt with: the goals and required verification levels are explained in the standard, and there are guidelines for the use of tools that automate developers' tasks.

In addition to the DO-178 (currently, version B) regulations, each airplane manufacturer usually has its own internal constraints: available hardware, delivery schedule, additional safety constraints, etc. Additionally, as programs tend to get larger and more complex, there is a permanent desire to use optimally the available hardware. Such a need is not necessarily in line with the aforementioned constraints: indeed, meeting them both is usually very challenging because performance and safety may be contradictory goals.

This paper describes the activities and challenges in an Airbus experiment that ultimately seeks to improve the performance of flight control software without reducing the level of confidence obtained by the development and verification strategy currently used. This experiment is carried around a very sensitive step in software development: assembly code generation from algorithmic language. A compiler may have a strong influence on software performance, as advanced compilers are able to generate optimized assembly code and such optimizations may be welcome if, for some reason, the source code is not itself optimal – in high level programming languages, the source code is unlikely to be optimal with respect to low level memory management (especially register and cache management). This work presents the performance-related analyses that were carried out to assess the interest of using an optimizing, formally-proved compiler, as well as the first ideas to make it suitable for application in certifiable software development.

The paper is structured as follows: Section 2 presents the fundamentals and challenges in the development of flight control software, and describes the methods used in this work to evaluate software performance, as well as the elements that weigh most in this aspect. Section 3 presents the CompCert compiler, the results of its performance evaluation and some ideas to use it confidently in such critical software. Section 4 draws conclusions from the current state of this work.

2 Flight Control Software and Performance Issues

2.1 An Overview of Flight Control Software

Since the introduction of the A320, Airbus relies on digital electrical flight control systems ("fly-by-wire") in its aircraft [2]. While older airplanes had only mechanical, direct links between the pilots' inputs and their actuators, modern aircraft rely on computers and electric connections to transmit these inputs. The flight control computers contain software that implement flight control laws, thus easing pilots' tasks – for example, a "flight envelope protection" is implemented not to let aircraft attain combinations of conditions (such as speed and G-load [2]) that are out of their specified physical limits and could cause failures.

It is clear that the dependability of such a system is tightly coupled with the dependability of its software, and the high criticality of a flight control system implies an equally high criticality of its software. As a result, flight control software are subject to the strictest recommendations (Software Level A) of the DO-178 standard: in addition to very rigorous planning, development and verification, there are "independence" guidelines (the verification shall not be done by the coding team) and the result of every automated tool used in the software development process is also subject to verification whenever it is used. These systematic tool output verification activities can be skipped if the tool is "qualified" to be used in a given software project. Tool qualification follows an approach similar to the certification of a flight software itself, as its main goal is to show that the tool is properly developed and verified, thus being considered as adequate for the whole software certification

process. The DO-178B makes a distinction between development and verification tools; development tools are those which may directly introduce errors in a program – such as a code generator, or a compiler – whereas verification tools do not have direct interference over the program, although their failure may also cause problems such as incorrect assumptions about the program behavior. The qualification of a development tool is much more laborious and requires a level of planning, documentation, development and verification that can be compared to the flight control software itself.

The software and hardware used in this work are similar to those described in [10]. The application is specified in the graphical formalism SCADE, which is then translated to C code by a qualified automatic code generator. The C code is finally compiled and linked to produce an executable file. The relevant hardware in the scope of this work currently comprises the PowerPC G3 microprocessor (MPC755), its L1 cache memory and an external RAM memory. The MPC755 is a single-core, superscalar, pipelined microprocessor, which is much less complex than modern multi-core processors but contains enough resources not to have an easily predictable time behavior.

In order to meet DO-178B guidelines, many verification activities are carried out during the development phases. While the code generator itself (developed internally) is qualified as a development tool, the compiler[1] is purchased and its inner details are not mastered by the development team. As a result, its qualification cannot be conducted and its output must be verified. However, verifying the whole generated code would be prohibitively expensive and slow. Since the code is basically composed of many instances of a limited set of "symbols", such as mathematic operations, filters and delays, the simplest solution is to make the compiler generate constant code patterns for each symbol. This can be achieved by limiting the code generator and compiler optimizations, and the code verification may be accomplished by verifying the (not very numerous) expected code patterns for each symbol with the coverage level required by the DO-178B, and making sure every compiled symbol follows one of the expected patterns. Other activities (usually test-based) are also carried out to ensure code integration and functional correctness.

2.2 Estimating Software Performance

The DO-178B requires a worst-case execution time (WCET) analysis to ensure correctness and consistency of the source code. Hardware and software complexity make the search for an exact WCET nearly impossible; usually one computes a time which is proved higher than the actual WCET, but not much higher, in order to minimize resource waste - for software verification and certification means, the estimated/computed WCET must be interpreted as the actual one.

As explained by Souyris *et al* in [10], the earlier method of calculating the WCET of Airbus's automatically generated flight control software was essentially summing the execution times of small code snippets in their worst-case scenarios. The proofs that the estimated WCET was always higher than the actual one did not need to be formal, thanks to the simplicity of the processor and memory components available at that time - careful reviews were proved sufficient to ensure the accuracy of the estimations. On the other hand, modern microprocessors have several resources – such as cache memories, superscalar pipelines, branch prediction and instruction reordering – that accelerate their average performance but make their behavior much more complicated to analyze.

[1] For confidentiality reasons, the currently used compiler, linker and loader names are omitted.

While a WCET estimation that does not take these resources into account would make no sense, it is not feasible to make manual estimations of a program with such hardware complexity. The current approach at Airbus [10] relies on AbsInt[2]'s automated tool a[3] [5] (which had to be qualified as a verification tool) to compute the WCET via static code analysis. In order to obtain accurate results, the tool requires a precise model of the microprocessor and other influent components; this model was created during a cooperation between Airbus and AbsInt. In addition, sometimes it is useful (or even essential) to give a[3] some extra information about loop or register value bounds to refine its analysis. Examples of these "hints", which are provided in annotation files, are shown in [10]. As the code is generated automatically, an automatic annotation generator was devised to avoid manual activities and keep the efficiency of the development process. In order to minimize the need for code annotations, and to increase overall code safety, the symbol library was developed so as to be as deterministic as possible.

2.3 Searching for performance gains

In a process with so many constraints of variable nature, it is far from obvious to find practical ways to generate "faster" software: the impact of every improvement attempt must be carefully evaluated in the process as a whole - a slight change in the way of specifying the software may have unforeseen consequences not only in the code, but even in the highest-level verification activities. It is useful to look at the V development cycle (which is advocated by the DO-178B) so as to find what phases may have the most promising improvements:

- Specification: Normally, the specification team is a customer of the development team. Specification improvements may be discussed between the two parts, but they are not directly modifiable by the developers.
- Design: In an automatic code generation process, the design phase becomes a part of the specification and is thus out of the development team scope.
- Coding: The coding phase is clearly important for the software performance. In the pattern coding level, there are usually few improvements to be made: after years of using and improving a pattern library, finding even more optimizations is difficult and time-consuming. However, the code generators and the compilers may be improved by relaxing this pattern-based approach in the final library code.
- Verification: In the long run, one must keep an eye on the new verification techniques that arise, because every performance gain is visible only if the WCET estimation methods are accurate enough to take them into account – sub-optimal specification and coding choices might have been made due to a lack of strong verification techniques at one time.

This work presents the current state of some experiments that are being performed in order to improve the compilation process.

3 A new approach for compiler verification

3.1 Qualification constraints for a compiler

The DO-178B states that a compiler is deemed acceptable when the overall software verification is successfully carried out. Specific considerations with respect to compilers include:

- Compiler optimizations do not need to be verified if the software verification provides enough coverage for the given criticality level.

[2] www.absint.com

- Object code that is not directly traceable to source code must be detected and verified with adequate coverage.

Thus, an optimizing compiler must be qualified, or additional verification activities must be carried out to ensure traceability and compliance of the object code.

Section 2.1 states that the trust in a development process that includes a "black-box" compiler is achieved by banning all compiler optimizations in order to have a simple structural traceability between source and binary code patterns. Traceability is used to attain Multiple Condition Decision Coverage (MC/DC) over the code structure of each symbol of the library. The coverage of the whole automatically-generated code is ensured, as it is a concatenation of such separately tested patterns. Other goals are also achieved with predictable code patterns:

- It is possible to know exactly what assembly code lines of the automatically-generated code require annotations to be correctly analyzed by a[3], as there are relatively few library symbols that require annotations, each one with just a few possible patterns.
- Compiler analyses can be done automatically, as its correctness is established by a simple code inspection: every generated pattern for a given symbol must match one of the unit-tested patterns for the same symbol. Compiler, assembler and linker are also tested during the integration tests: as the object code is executed on the actual target computer, the DO-178B code compliance requirements would not be fulfilled if there were wrong code or mapping directives.

Thus, several objectives are accomplished with a non-optimized code, and a different approach would lead to many verification challenges. COTS compilers usually do not provide enough information to ensure their correctness, especially when taking optimizations into account. If developers could actually master a compiler behavior, the DO-178B tool qualification might give way to a more flexible (albeit laborious) way of compiling.

3.2 CompCert: Towards a trusted compiler

One can figure out that traditional COTS (Commercial off-the-shelf) compilers are not adapted to the rigorous development of flight control software – the notion of "validated by experience" tool is not acceptable for highly critical software development tools. However, there have been some advances in the development of compilers, with interesting works that discuss the use of formal methods to implement "correct" compilers[3], either by verifying the results of their compilation [7] or by verifying the compiler semantics [12, 6]. In the scope of this work, a most promising development is the CompCert[4] compiler. Its proved subset is broader in comparison to other experimental compilers, it compiles most of the C language (which is extensively used in embedded systems), and it can generate Assembly code for the MPC755.

As explained in [6], CompCert is mostly programmed and proved in Coq, using multiple phases to perform an optimized compilation. Its optimizations are not very aggressive, though: as the compiler's main purpose is to be "trustworthy", it carries out basic optimizations such as constant propagation, common subexpression elimination and register allocation by graph coloring, but no loop optimizations, for instance. As no code optimizations are enabled in the currently used compiler, using a few essential optimization options could already give good performance results.

[3] In this work, the term "certifying compilation", found in previous works such as [7], is not used in order to avoid confusion with avionics software certification.

[4] `http://compcert.inria.fr`

3.3 Performance evaluation of CompCert

In order to carry out a meaningful performance evaluation, the compiler was tested on a prototype as close as possible to an actual flight control software. As this prototype has its own particularities with relation to compiler and mapping directives, some adaptations were necessary in both the compiler and the code. To expedite this evaluation, CompCert was used only to generate assembly code for the application, while the "operational system" was compiled with the default compiler. Assembling and linking were also performed with the default tools, for the same reason. Figure 1 illustrates the software development chain.

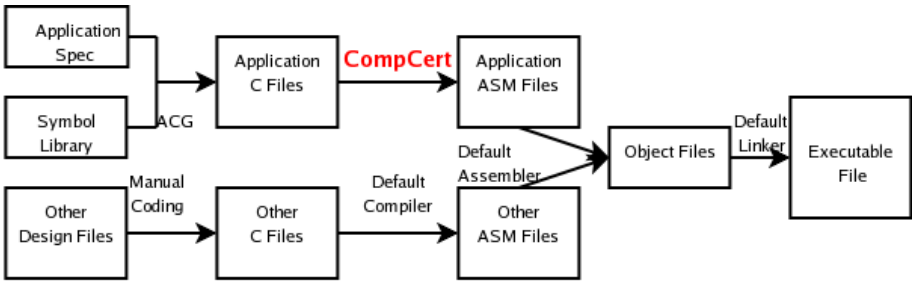


Figure 1 The development chain of the analyzed program

About 2500 files (2.6MB of assembly code with the currently used compiler) were compiled with CompCert (version 1.7.1-dev1336) and with three configurations of the default compiler: non-optimized, optimized without register allocation optimizations, and fully optimized. A quick glance at some CompCert generated code was sufficient to notice interesting changes: the total code size is about 26% smaller than the code generated by the default compiler. This significant improvement has its roots in the specification formalism itself: a potentially long sequential code is composed by a sequence of mostly small symbols, each one with its own inputs and outputs. Thus, a non-optimizing compiler must do all the theoretically needed load and store operations for each symbol. For traceability purposes, the register allocation is done manually for the non-optimized code and CompCert manages to generate more compact Assembly code by ignoring the user-defined register allocation. Listing 1 depicts a non-optimized simple symbol that computes the sum of two floating-point numbers. As this symbol is often in sequence with other symbols, it is likely that its inputs were computed just before and its output will be used in one of the next scheduled instructions. If there are enough free registers, CompCert will simply keep these variables inside registers and only the `fadd` instruction will remain, as shown in Listing 2.

Listing 1 Example of a symbol code

```
lfd f3, 8(r1)
lfd f4, 16(r1)
fadd f5, f4, f3
stfd f5, 24(r1)
```

Listing 2 Its optimized version

```
fadd f5, f4, f3
```

As the local variables are usually kept on a stack located in the cache, analyses showed that CompCert generates code with about 76% fewer cache reads and 65% fewer cache writes. Table 1 compares these results with those of the default compiler in optimized configurations, with the default non-optimized code as the reference.

In order to see the effects of this code size reduction, a[3] was used to compute the WCET for all analyzed nodes – we do not seek interprocedural optimizations or a register allocation

	Code Size	Cache Reads	Cache Writes
CompCert	-25.7%	-76.4%	-65.1%
Default (optimized without register allocation)	+0.8%	+19.9%	+23.4%
Default (fully optimized)	-38.2%	-81.8%	-76.6%

Table 1 Code size and memory access comparison

that goes beyond one single node, hence individual WCET computations are meaningful in this context. The results are encouraging: the mean of the WCET of the CompCert compiled code was 12.0% lower than the reference. Without register allocation, the default compiler presented a reduction of only 0.5% in WCET, while there was a reduction of 18.4% in the WCET of the fully optimized code. The WCET comparison for each of the analyzed nodes is depicted in Figure 2. The WCET improvement is not constant over all nodes: some of

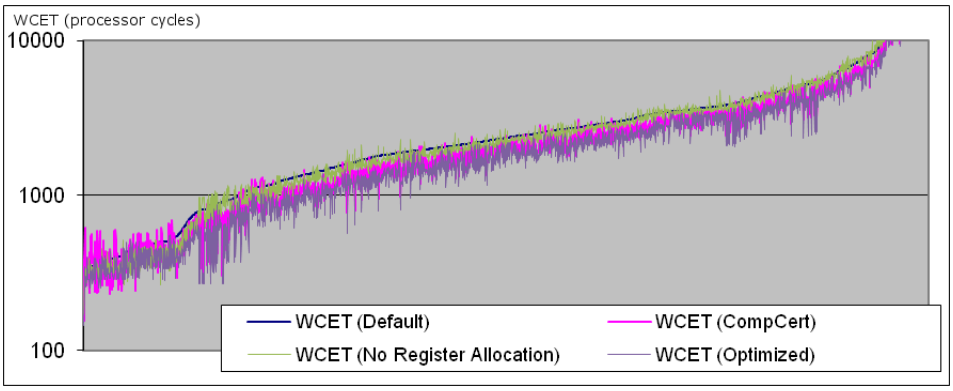


Figure 2 WCET for all analyzed program nodes

them do not have many instructions, but they do have strong performance "bottlenecks" such as hardware signal acquisitions, which take considerable amounts of time and are not improved by code optimization. In addition, CompCert's recent support for small data areas was not used in the evaluation, while it is used by the default compiler. Nonetheless, the overall WCET is clearly lower.

The results of these WCET analyses emphasizes the importance of a good register allocation and how other optimizations are hampered without it.

3.4 Generating annotations for WCET analysis

As mentioned in Section 2.2, annotations over automatically-generated code are mandatory to increase the WCET analysis precision whenever an accessed memory address or a loop guard depends on the value of a floating-point variable, or a static variable that is not updated inside the analyzed code. We have prototyped a minor extension to the CompCert compiler that supports writing annotations in C code, transmitting them along the compilation process, and communicating them to the a^3 analyzer. The input language of CompCert is extended with the following special form:

```
__builtin_annotation("0 <= %1 <= %2 < 360", i, j);
```

which looks like a function call taking a string literal as first argument and zero, one or more C variables as extra arguments. Semantically and throughout the compiler, this special form

is treated as a *pro forma* effect, as if it were to print out the string and the values of its arguments when executed. CompCert's proof of semantic preservation therefore guarantees that control flows through these annotation statements at exactly the same instants in the source and compiled code, and that the variable arguments have exactly the same numerical values in both codes. At the very end of the compilation process, when assembly code is printed, no machine instructions are generated for annotation statements. Instead, a special comment is emitted in the assembly output, consisting of the string argument (`"0 <= %1 <= %2 < 360"` in the example above) where the `%`*i* tokens are substituted by the final location (machine register, stack slot or global symbol) of the *i*-th variable argument. For instance, we would obtain "`# annotation: 0 <= r3 <= @32 < 360`" if the compiler assigned register `r3` to variable `i` and the stack location at stack pointer plus 32 bytes to variable `j`. The listing generated by the assembler then shows this comment and the program counter (relative to the enclosing function) where it occurs. From this information, a suitable annotation file can be automatically generated for use by the a^3 analyzer.

Several variants on this transmission scheme can be considered, and the details are not yet worked out nor experimentally evaluated. Nonetheless, we believe that this general approach of annotating C code and compiling these annotations as *pro forma* effects is a good starting point for the automatic generation of annotations usable during WCET analysis.

3.5 CompCert and the avionics software context

After the successful performance evaluation, the feasibility of the use of CompCert in an actual flight control software development must be studied more thoroughly. Given all the constraints and regulations explained in this paper, this task will take a significant amount of time, as all constraints from several actors (customers, development, verification, certification) must be taken into account.

When an automatic code generator is used, it is clear that the customers want a highly reactive development team. A million-line program (with a great deal of its code being generated automatically) must be coded and verified in a few days; with such a strict schedule, little or no manual activities are allowed.

The development team also has its rules, in order to enforce correct methods and increase development safety. Thus, the compiler must generate a code that complies to an application binary interface (in this case, the PowerPC EABI) and other standards, such as IEEE754 for floating-point operations. Although this work used two compilers to build the whole software, CompCert will have to deal with all the program parts (the ACG-generated code is much bigger, but also simpler than the rest); it will also have to do assembling and linking.

The verification phase will be significantly impacted, given all the assumptions that were based on a code with predictable patterns:

Unit verification The unit verification of each library symbol will have to be adapted. With no constant code patterns, there is no way to attain the desired structural coverage by testing only a number of code patterns beforehand that then appears in sequence in the generated software. It would be too onerous to test the whole code after every compilation. A possible solution is to separate the verification activities of the source and object code. The verification of the source code can be done using formal methods, using tools that are already familiar inside Airbus, such as Caveat[5] and Frama-C [6] [11].

[5] `http://www-list.cea.fr/labos/gb/LSL/caveat/index.html`
[6] `http://frama-c.com`

Object code compliance and traceability can be accomplished using the formal proofs of the compiler itself, as they intend to ensure a correct object code generation. In this case, only one object code pattern needs to be verified (e.g. by unit testing) for each library symbol and the test results can be generalized for all other patterns, thanks to the CompCert correctness proofs.

WCET computation A new automatic annotation generator will have to be developed, as the current one relies on constant code patterns to annotate the code. The new generator will rely on information provided directly by CompCert (Section 3.4) to correctly annotate the code when needed.

Compiler verification It is clear that the CompCert formal proofs shall form the backbone of a new verification strategy. An important point of discussion is how these proofs can be used in an avionics software certification process. The most direct approach is qualifying the compiler itself as a development tool, but it is far from a trivial process: the qualification of a development tool is very arduous, and qualifying a compiler is a new approach that will require intensive efforts to earn the trust of certification authorities. Thus, CompCert has to meet DO-178B level A standards for planning, development, verification and documentation, and these standards largely surpass the usual level of safety achieved by traditional compiler development processes. An alternative method of verification, which is also being discussed, is using its correctness proofs in complementary (and automatic) analyses that will not go in the direction of qualifying CompCert as a whole, but should be sufficiently well-thought-out to prove that it did a correct compilation.

4 Conclusions and Future Work

This paper described a direction to improve performance for flight control software, given their large number of development and certification constraints. The motivation for using a formally proved compiler is straightforward: certifying a COTS compiler to operate without restrictions (such as hindering every possible code optimization) would be extremely hard, if not impossible, as information related to its development are not available. While the largest part of the work – the development of an appropriate development and verification strategy to work with CompCert – has just started, the performance results are rather promising. It became clear that the "symbol library" automatic code generation strategy implies an overhead in load and store operations, and a good register allocation can mitigate this overhead.

Future work with CompCert include its adaptation to the whole flight control software and the completion of the automated mechanism to provide useful information that can help in the generation of code annotations. Also, discussions among development, verification and certification teams in Airbus are taking place to study the needed modifications throughout the development process in order to use CompCert in a development cycle at least as safe as the current one. Parallel studies are being carried out to find new alternatives for software verification, such as Astrée [3], and evaluate their application in the current development cycle [11].

Another direction for future work is to further improve WCET by deploying additional optimizations in CompCert and proving that they preserve semantics. The WCC project of Falk *et al* [4] provides many examples of profitable WCET-aware optimizations, often guided by the results of WCET analysis. Proving directly the correctness of these optimizations appears difficult. However, equivalent semantic preservation guarantees can be achieved at lower proof costs by *verified translation validation*, whereas each run of a non-verified

optimization is verified *a posteriori* by a validator that is proved correct once and for all. For example, Tristan and Leroy [13] show a verified validator for trace scheduling (instruction scheduling over extended basic blocks) that could probably be adapted to handle WCC's superblock optimizations. Rival has experimented the translation validation approach on a wider scope in [8] but, currently, the qualification and industrialization of such a tool seems more complex.

In addition, the search for improvements in flight control software performance is not limited to the compilation phase. The qualified code generator is also subject to many constraints that limit its ability to generate efficient code. Airbus is already carrying out experiments in order to study new alternatives, such as the Gene-Auto project [9].

References

1 DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1982.

2 Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems. In *FTCS*, pages 616–623, 1993.

3 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In Mitsu Okada and Ichiro Satoh, editors, *ASIAN*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

4 Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *The International Journal of Time-Critical Computing Systems (Real-Time Systems)*, 46(2):251–300, 2010.

5 Reinhold Heckmann and Christian Ferdinand. Worst-case Execution Time Prediction by Static Program Analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*, pages 26–30. IEEE Computer Society, 2004.

6 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

7 George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. *SIGPLAN Not.*, 33(5):333–344, 1998.

8 Xavier Rival. Symbolic transfer functions-based approaches to certified compilation. In *31st Symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

9 Ana-Elena Rugina and Jean-Charles Dalbin. Experiences with the Gene-Auto Code Generator in the Aerospace Industry. In *Proceedings of the Embedded Real Time Software and Systems (ERTS^2)*, 2010.

10 Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, and Guillaume Borios. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

11 Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal Verification of Avionics Software Products. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546. Springer, 2009.

12 Martin Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77. Springer Verlag, 2002.

13 Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.