WCET TOOL CHALLENGE 2011: REPORT

Reinhard von Hanxleden[1], Niklas Holsti[2], Björn Lisper[3], Erhard Ploedereder[4], Reinhard Wilhelm[5] (Eds.), Armelle Bonenfant[6], Hugues Cassé[6], Sven Bünte[7], Wolfgang Fellger[4], Sebastian Gepperth[4], Jan Gustafsson[3], Benedikt Huber[7], Nazrul Mohammad Islam[3], Daniel Kästner[8], Raimund Kirner[9], Laura Kovács[7], Felix Krause[4], Marianne de Michiel[6], Mads Christian Olesen[10], Adrian Prantl[11], Wolfgang Puffitsch[7], Christine Rochange[6], Martin Schoeberl[12], Simon Wegener[8], Michael Zolda[7], Jakob Zwirchmayr[7]

Abstract

Following the successful WCET Tool Challenges in 2006 and 2008, the third event in this series was organized in 2011, again with support from the ARTIST DESIGN Network of Excellence. Following the practice established in the previous Challenges, the WCET Tool Challenge 2011 (WCC'11) defined two kinds of problems to be solved by the Challenge participants with their tools, WCET problems*, which ask for bounds on the execution time, and* flow-analysis problems*, which ask for bounds on the number of times certain parts of the code can be executed. The benchmarks to be used in WCC'11 were debie1, PapaBench, and an industrial-strength application from the automotive domain provided by Daimler. Two default execution platforms were suggested to the participants, the ARM7 as "simple target" and the MPC5553/5554 as a "complex target," but participants were free to use other platforms as well. Ten tools participated in WCC'11: aiT, Astrée, Bound-T, FORTAS, METAMOC, OTAWA, SWEET, TimeWeaver, TuBound and WCA.*

[1]Department of Computer Science, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24098 Kiel, Germany
[2]Tidorum Ltd, Tiirasaarentie 32, FI-00200 Helsinki, Finland
[3]School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
[4]Institute of Software Technology (ISTE), University of Stuttgart, Universitätsstr. 38, 71229 Stuttgart, Germany
[5]FR. 6.2 - Computer Science, Universität des Saarlandes, PO-Box 15 11 50, 66041 Saarbrücken
[6]IRIT - CNRS, Université de Toulouse, France
[7]Faculty of Informatics, Technical University Vienna, 1040 Vienna
[8]AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany
[9]Compiler Technology and Computer Architecture Group, University of Hertfordshire, Hatfield, Hertfordshire, AL10 9AB, UK
[10]Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg, Denmark
[11]Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94551, USA
[12]Department of Informatics and Mathematical Modeling, Technical University of Denmark, Asmussens Alle, DTU - Building 305, 2800 Lyngby, Denmark

1. Introduction

The chief characteristic of (hard) real-time computing is the requirement to complete the computation within a given time or by a given deadline. The computation or execution time usually depends to some extent on the input data and other variable conditions. It is then important to find the worst-case execution time (WCET) and verify that it is short enough to meet the deadlines in all cases.

1.1. The WCET Problem Statement

Several methods and tools for WCET analysis have been developed. Some tools are commercially available. The survey by Wilhelm et al. [42] is a good introduction to these methods and tools. Some tools use pure static analysis of the program; other tools combine static analysis with dynamic measurements of the execution times of program parts. Unlike most applications of program analysis, WCET tools must analyse the *machine code*, not (only) the source code. This means that the analysis depends on the target processor, so a WCET tool typically comes in several versions, one for each supported target processor or even for each target system with a particular set of caches and memory interfaces. Some parts of the machine-code analysis may also depend on the compiler that generates the machine code. For example, the analysis of control-flow in switch-case statements may be sensitive to the compiler's idiomatic use of jumps via tables of addresses.

In general, WCET tools use simplifying approximations and so determine an *upper bound* on the WCET, not the *true* WCET. The pessimism, that is the difference between the true WCET and the upper bound, may be large in some cases. For most real, non-trivial programs a fully automatic WCET analysis is not (yet) possible which means that manual *annotations* or *assertions* are needed to define essential information such as loop iteration bounds. The need for such annotations, and the form in which the annotations are written, depends on both the WCET tool and on the target program to be analysed.

1.2. The WCET Tool Challenge: Aims and History

As the term "Challenge" suggests, the aim is not to find a "winning" tool but to challenge the participating tools with common benchmark problems and to enable cross-tool comparisons along several dimensions, including the degree of analysis automation (of control-flow analysis, in particular), the expressiveness and usability of the annotation mechanism, and the precision and safety of the computed WCET bounds. Through the Challenge, tool developers can demonstrate what their tools can do, and potential users of these tools can compare the features of different tools.

Jan Gustafsson of the Mälardalen Real-Time Centre organized the first WCET Tool Challenge in 2006, using the Mälardalen benchmark collection [10] and the PapaBench benchmark [31], with participation from five tools (aiT, Bound-T, SWEET, MTime, and Chronos). The results from WCC'06 were initially reported at the ISoLA 2006 conference [8] and later in complete form [9]. Lili Tan of the University of Duisburg-Essen did an independent evaluation of the tools on these benchmarks, also reported at ISoLA 2006 [40].

The second WCET Tool Challenge was organized in 2008 (WCC'08) and was presented at the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis [13]. Two of the WCC'06 participants (Bound-T and MTime) as well as four new tools (OTAWA, RapiTime, TuBound and wcc) participated in WCC'08. The second Challenge differed from the first in that it suggested a common

Tool	Description in Section	Source-code flow analysis	ARM7 (Sec. 2.4.1)	MPC5554 (Sec. 2.4.2)	Other target processors
aiT	3.1		+	+	
Astrée	3.2	+			
Bound-T	3.3		+		
FORTAS	3.4	+			TC1796 (Sec. 2.4.3)
METAMOC	3.5				
OTAWA	3.6	+	+	+	
SWEET	3.7	+			
TimeWeaver	3.8			+	
TuBound	3.9	+			C167 (Sec. 2.4.4)
WCA	3.10				JOP (Sec. 2.4.5)

Table 1: Participating tools and target processors used in WCC'11.

execution platform (the ARM7 LPC2138) and also defined pure flow-analysis problems. It included less benchmarks (5 instead of 17), but increased the number of analysis problems.

1.3. Contributions and Outline

The third WCET Tool Challenge was organized in 2011 (WCC'11[2]) and is the subject of this report. As a first overview, Table 1 lists the tools participating in the Challenge and indicates which target processors each participant has addressed for WCC'11; most tools support other target processors, too.

This report combines contributions from the WCC'11 participants and is edited by the WCC'11 steering group, some of whom are also WCC'11 participants. Sec. 2 describes the organization of the Challenge. This is followed by Sec. 3, the most substantial section of this report, where the participants in turn describe their tools and the experiences in participating in the Challenge. The overall results are reported in Sec. 4. In addition to the tool authors who tested their tools on the debie1 and PapaBench benchmarks, a group of students of the University of Stuttgart, led by Erhard Ploedereder, tried some of the tools on a proprietary benchmark supplied by Daimler; they report their experience in Sec. 5. The paper concludes in Sec. 6.

It should be noted that not only the Challenge itself, but also this report adopts much from the previous edition, including both structure and content. Specifically, the WCET problem statement, the descriptions of the ARM processor and the debie1 benchmark, and the presentation of the types of analysis problems are largely cited from the WCC'08 report [13]. Maintaining most of the report structure may facilitate tracing the development of the Challenge and the participating tools. What we did change was to move the tool description together with the reports from the tools. Also, we decided to fold the "Problems and Solutions" section into the section on organization, as we felt that this time, there were not enough general problems (apart from the experiences with the individual tools) to report on that would justify their own section.

[2] `http://www.mrtc.mdh.se/projects/WCC/` — this web page links to the wiki of the WCC'11 as well as to the previous WCC editions.

2. Organization of WCC'11

After WCC'08, a working group for the next Challenge was set up. Upon the original initiative by Reinhard Wilhelm, Erhard Ploedereder offered to lead a group of students to serve as evaluators—funded by ARTIST—on a real-world benchmark supplied by Daimler. Björn Lisper—responsible for the Timing Analysis activity within the ARTIST DESIGN Network of Excellence on embedded systems design—joined in and volunteered to organize the wiki. Niklas Holsti, who could also draw on his experience in running WCC'08, and Reinhard von Hanxleden, who agreed to serve as chair, completed the WCC'11 Steering Committee.

As in the previous edition of the Challenge, the main tools for running the Challenge were the mailing list and the wiki. As a change from last time, results were not filled into the wiki, but instead were collected in simple comma-separated value (csv) files following a predefined format (tool name, benchmark name, target processor name, target cross-compiler name, analysis problem name, analysis question name, analysis result, possible remarks). This turned out to be a very convenient way to consolidate the results.

2.1. WCC'11 Schedule, Problems and Solutions

The goal formulated early on was to present the results of the Challenge at the WCET'11 workshop, which put a firm boundary on the overall schedule. Planning for WCC'11 began in earnest in September 2010. Potential participants were gathered in a mailing list, and gave feedback on potential target architectures in October (see Sec. 2.4). In November, the benchmarks were selected (Sec. 2.2). In December, the Wiki was launched. The official, virtual kick-off of the Challenge took place on January 28, 2011. Result reports from the participants were requested by May 27.

Once the "rules of the game" were clear and the Challenge was under way, there were only little difficulties that were escalated to the mailing list and/or the Steering Committee. As already mentioned in the introduction, this is reflected in the lack of a dedicated "Problems and Solutions" section in this report. There were some clarifications needed for some of the analysis problems and for the memory access timing of the ARM7; a port (compilation and binary) to the MPC5554 target was contributed by Simon Wegener for the debie1 benchmark; for PapaBench, analysis problems and an ARM7 binary were provided by Hugues Cassé.

2.2. The Benchmarks

Thanks to Daimler, we could from the beginning count on an industrial-size, real-world benchmark to be included in WCC'11, see Sec. 2.2.3. However, it was also clear rather early that this benchmark would not be suitable for all analysis tools. To broaden the tool base, and to also be attractive to participants of previous Challenges, we decided to reuse two benchmarks, the PapaBench already used in WCC'06 (Sec. 2.2.2), and the debie1 benchmark introduced in WCC'08 (see Sec. 2.2.1).

2.2.1. The debie1 benchmark

The debie1 (First Standard Space Debris Monitoring Instrument, European Space Agency[3]) benchmark is based on the on-board software of the DEBIE-1 satellite instrument for measuring impacts

[3] `http://gate.etamax.de/edid/publicaccess/debie1.php`

of small space debris and micro-meteoroids. The software is written in C, originally for the 8051 processor architecture, specifically an 80C32 processor that is the core of the the Data Processing Unit (DPU) in DEBIE-1. The software consists of six tasks (threads). The main function is interrupt-driven: when an impact is recorded by a sensor unit, the interrupt handler starts a chain of actions that read the electrical and mechanical sensors, classify the impact according to certain quantitative criteria, and store the data in the SRAM memory. These actions have hard real-time deadlines that come from the electrical characteristics (hold time) of the sensors. Some of the actions are done in the interrupt handler, some in an ordinary task that is activated by a message from the interrupt handler. Two other interrupts drive communication tasks: telecommand reception and telemetry transmission. A periodic housekeeping task monitors the system by measuring voltages and temperatures and checking them against normal limits, and by other checks. The DEBIE-1 software and its WCET analysis with Bound-T were described at the DASIA'2000 conference [14]. The WCC'11 Wiki also has a more detailed description.

The real DEBIE-1 flight software was converted into the debie1 benchmark by removing the proprietary real-time kernel and the low-level peripheral interface code and substituting a test harness that simulates some of those functions. Moreover, a suite of tests was created in the form of a test driver function. The benchmark program is single-threaded, not concurrent; the test driver simulates concurrency by invoking thread main functions in a specific order. The DEBIE-1 application functions, the test harness, and the test driver are linked into the same executable. This work was done at Tidorum Ltd by Niklas Holsti with ARTIST2 funding.

Space Systems Finland Ltd (SSF), the developer of the DEBIE-1 software, provides the software for use as a WCET benchmark under specific Terms of Use that do not allow fully open distribution. Therefore, the debie1 benchmark is not directly down-loadable from the WCC'11 Wiki. Copies of the software can be requested from Tidorum[4]. SSF has authorized Tidorum to distribute the software for such purposes.

2.2.2. The PapaBench Benchmark

PapaBench [31] is a WCET benchmark derived from the Paparazzi UAV controller. This controller has been developed in the ENAC school in Toulouse and targets low-cost UAV, that is, model airplane embedding a microprocessor. This controller has been successful to drive a lot of different models in complex autonomous missions and has won several awards in this domain.

Basically, the UAV is made of several actuators (motor, flaps, etc) and a very light set of sensors including a GPS (connected by a serial port) and an infrared sensor to control slope. The system may be controlled from ground using a classical wireless link or may fly in an autonomous mode performing a pre-programmed mission. In this case, the wireless descending link is only used to transfer flight log or video if the payload is composed of a little camera.

In its original configuration, the computing hardware was composed of two ATMEL AVR microprocessors communicating by a SPI link. The first one, fbw (fly-by-wire), was responsible for the control of actuators and sensors and for the stabilization of the flight. It was also used to perform commands emitted by the wireless link. The second microprocessor, autopilot, was a bit more powerful and was concerned with the realization of the mission, that is, the choice of the flight plan. The system has several emergency modes activated according to the whole system state. In a first mode, it tries to

[4] `http://www.tidorum.fi/`

return to its "home" base. In another one, it tries to save the integrity of the model plane by ensuring a minimal landing drive. And in a very bad case, it puts the actuators in a configuration ensuring it will simply plane gracefully in the hope it may land without breaking anything.

To perform a flight, the first task is to program a flight plan and to generate automatically a piece of code included in the embedded software system. Then, the full system is compiled and composed of two binary programs: fbw and autopilot. In the next step, the programs are transferred to the controller and the plane is launched (by hand) and the controller starts to drive the plane. If all is ok, the flight plan ends with the model plane landing at its starting point.

2.2.3. The Daimler benchmark

The benchmark is part of a control system for trucks that deals with, among others, collision detection. The code is compiled for the MPC 5553 architecture using the WindRiver Diab compiler. The target processor does not have any external memory. VLE instructions are not used.

Analysis problems were available. Due in part to circumstances described in section 5.2, the ultimate choice of WCET questions was directed at four entry points below the task level of different types:

- An interrupt handler `INTERR`

  This was basically a simple test to get the tools to work and to familiarize the students with them. It is a simple interrupt handler that only calls one function and does not include any loops.

- An initialization routine `INIT`

  This is a second simple entry point that sets some variables, does not call any functions and has no loops.

- Two calculation routines `CALC1` and `CALC2`

  These routines execute moderately complex numeric calculations. They include some loops and static function calls.

- A complete task of the embedded system `TASK`

  This is a typical task of an embedded system; it is the body of an endless loop that executes some subtasks and then suspends itself until it needs to run again.

2.3. The Analysis Problems

For each WCC benchmark, a number of analysis problems or questions are defined for the participants to analyse and answer. There are two kinds of problems: WCET-analysis problems and flow-analysis problems. Flow-analysis problems can be answered by tools that focus on flow-analysis (for example SWEET) but that do not have the "low-level" analysis for computing WCET bounds (for the ARM7 processor, or for any processor). Flow-analysis problems can also show differences in the flow-analyses of different WCET tools, and this may explain differences in the WCET bounds computed by the tools.

A typical WCET-analysis problem asks for bounds on the WCET of a specific subprogram within the benchmark program (including the execution of other subprograms called from this subprogram). For example, problem 4a-T1 for the debie1 benchmark asks for the WCET of the Handle_Telecommand function when the variable input data satisfy some specific constraints.

A typical flow-analysis problem asks for bounds on the number of times the benchmark program executes a certain statement, or a certain set of statements, within one execution of a root subprogram. For example, problem 4a-F1 for the debie1 benchmark asks how many calls of the macro `SET_DATA_BYTE` can be executed within one execution of the function Handle-Tele-command, under the same input-data constraints as in the WCET-analysis problem 4a-T1. By further requiring the analysis to assume that the execution time of `SET_DATA_BYTE` is arbitrarily large we make it possible for pure WCET-analysis tools to answer this flow-analysis question, since this assumption forces the worst-case path to include the maximum number of `SET_DATA_BYTE` calls; all alternative paths have a smaller execution time.

2.4. The Target Processors

After polling the potential participants, we decided to suggest two common target processors for WCC'11, a "simple" processor and a "complex" processor. However, participants were welcome to use other processors as well.

2.4.1. The "Simple" Processor: ARM7

For the "simple" processor, we chose to select the same processor as was already used for WCC'08, the ARM7, as e. g. on the LPC2138 board from NXP Semiconductor. As has been elaborated in the report on WCC'08 [13], the ARM7 also offers a MAM (Memory Acceleration Module), which, however, significantly complicates the timing analysis. We therefore suggested to de-activate the MAM. The following is a brief description of the ARM7, based on the WCC'08 report.

The ARM7 [1] is basically a simple, deterministic processor that does not challenge the analysis of caches and complex pipelines that are important features of some WCET tools [42]. The ARM7 is a 32-bit pipelined RISC architecture with a single (von Neumann) address space. All basic ARM7 instructions are 32 bits long. Some ARM7 devices support the alternative THUMB instruction set, with 16-bit instructions, but this was not used in WCC'11. The ARM7 processor has 16 general registers of 32 bits. Register 15 is the Program Counter. Thus, when this register is used as a source operand it has a static value, and if it is a destination operand the instruction acts as a branch. Register 14 is designated as the "link register" to hold the return address when a subprogram call occurs. There are no specific call/return instructions; any instruction sequence that has the desired effect can be used. This makes it harder for static analysis to detect call points and return points in ARM7 machine code. The timing of ARM7 instructions is basically deterministic. Each instruction is documented as taking a certain number of "incremental" execution cycles of three kinds: "sequential" and "non-sequential" memory-access cycles and "internal" processor cycles. The actual duration of a memory-access cycle can depend on the memory subsystem. The term "incremental" refers to the pipelining of instructions, but the pipeline is a simple linear one, and the total execution-time of an instruction sequence is generally the sum of the incremental times of the instructions.

The LPC2138 chip The NXP LPC2138 implements the ARM7 architecture as a microcontroller with 512 KiB of on-chip flash memory starting at address zero and usually storing code, and 32 KiB

of static on-chip random-access memory (SRAM) starting at address 0x4000 0000 and usually storing variable data. There is no off-chip memory interface, only peripheral I/O (including, however, I2C, SPI, and SSP serial interfaces that can drive memory units).

The on-chip SRAM has a single-cycle (no-wait) access time at any clock frequency. The on-chip flash allows single-cycle access only up to 20 MHz clock frequency. At higher clock frequencies, up to the LPC2138 maximum of 60 MHz, the flash needs wait cycles. This can delay instruction fetching and other flash-data access. The LPC2138 contains the aforementioned device called the Memory Acceleration Module (MAM) that reduces this delay by a combination of caching and prefetching; however, as already mentioned, we suggested to de-activate the MAM.

The on-chip peripherals in the LPC2138 connect to a VLSI Peripheral Bus (VPB) which connects to the Advanced High-performance Bus (AHB) through an AHB-VPB bridge. This bus hierarchy causes some delay when the ARM7 core accesses a peripheral register through the AHB. If the VPB is configured to run at a lower clock frequency than the ARM7 core this delay is variable because it depends on the phase of the VPB clock when the access occurs.

The programming tools The IF-DEV-LPC kit from iSYSTEM came with an integrated development environment called WinIDEA and a GNU cross-compiler and linker. The distributed benchmark binaries for WCC'11 were created with Build 118 of these tools using gcc-4.2.2[5]. The IF-DEV-LPC kit has an USB connection to the controlling PC and internally uses JTAG to access the LPC2138. WinIDEA supports debugging with breakpoints, memory inspections, and so on.

2.4.2. The "Complex" Processor: MPC5553/5554

The Freescale MPC5553/MPC5554 micro-controllers implement the PowerPC Book E instruction set. The Book E instruction set adapts the normal PowerPC ISA to the special needs of embedded systems. The normal floating point instructions are replaced by digital signal processing instructions.

Both micro-controllers have a two-level memory hierarchy. They use a unified cache (8 KB on the MPC5553, 32 KB on the MPC5554) to accelerate the accesses to the internal SRAM and Flash memory. Additionally, they support the use of external memory. The memory management unit has a 32-entry translation look-aside buffer. The load/store subsystem is fully pipelined and an 8-entry store buffer is used to accelerate the instruction throughput.

The unified cache is 2-way set associative on the MPC5553 and 8-way set associative on the MPC5554. The cache can be locked on a per way basis. Moreover, a way can be declared as instruction or data cache only. As another acceleration mechanism, the micro-controllers support branch prediction. The processors run at a clock speed of up to 132 MHz.

Various peripherals can be attached to the micro-controllers, for example by using the FlexCAN bus. The MPC55xx micro-controllers support debugging through the IEEE-ISTO 5001-2003 NEXUS interface and the IEEE 1149.1 JTAG controller.

[5] `http://www.isystem.si/SWUpdates/Setup_IFDEV_9_7_118/iFDEVSetup.exe`

2.4.3. The TriCore 1796

The TriCore 1796 and the TriBoard TC1796 were the chosen target of the FORTAS tool (see Sec. 3.4). The TC1796 is based on the 32-bit TriCore 1.3 load/store architecture. In the following description we focus on the features that we consider particularly relevant for execution timing and measurement. For details, please refer to the processor manual [23].

The TC1796 uses a Harvard architecture with separate buses to program and data memory, i.e., instruction fetching can be performed in parallel with data accesses. The 4GB address space is partitioned into 16 equally-sized segments. For the challenge, program code was stored in segment 8, which provides cached memory accesses via the *External Bus Unit* (EBU). The instruction cache is characterized by the following features:

- Two-way set-associativity
- LRU replacement strategy
- Line size of 256 bits
- Cache can be globally invalidated
- Cache can be globally bypassed
- Unaligned accesses crossing caches line supported with a penalty of 1 CPU cycle

There is no data cache, but all data written by ST (store) or LDMST (load-modify-store) instructions is buffered. The buffer content is written to memory when the CPU and the Data Local Memory Bus are both idle.

Execution timing is also affected by the superscalar design. The TC1796 has a top-level pipeline consisting of an *Instruction Fetch Unit*, an *Execution Unit* and a *General Purpose Register File*. Within the execution unit the pipeline splits into three parallel sub-pipelines: an *Integer Pipeline*, which mainly handles data arithmetics and conditional jumps, a *Load Store Pipeline*, which is mainly responsible for memory accesses, unconditional jumps, calls and context switching, and a *Loop Pipeline*, which mainly handles special loop instructions providing zero-overhead loops. Consequently, up to three instructions can be issued and executed in parallel. Also, a floating point unit is attached to the CPU as a coprocessor.

Furthermore, there is a static branch predictor that implements the following rules [20]:

- Backward and short forward branches (16-bit branches with positive displacement) are predicted taken
- Non-short forward branches are predicted not taken

The overhead of the different cases is summarized in Table 2.

The TC1796 offers *On-Chip Debug Support* (OCDS) *Level 1* and *Level 2* for debugging and execution time measurement. OCDS Level 1 includes a JTAG module, which can be used to download programs

Prediction	Outcome	Penalty (cycles)
not taken	not taken	1
not taken	taken	3
taken	not taken	3
taken	taken	2

Table 2: Branch penalties.

to the target and to inject input data. Tracing is enabled via OCDS Level 2, a vendor-specific variant of the *Nexus IEEE-ISTO 5001-2003* standard interface[6]. For the challenge, this interface was used to sample time-stamped program flow information at each CPU cycle without exerting a probing effect. Code instrumentation is not necessary.

Target Platform: TriBoard TC1796 In the following, we focus on those features that we consider particularly relevant for execution timing and measurement. For details, please refer to the board manual [22].

The TriBoard is equipped with 4MB of Burst Flash memory and 1 MB of asynchronous SRAM, which are both connected to the processing core via the External Bus Unit of the processor, and these are the only devices that are connected to the EBU. For the challenge, both program data and program instructions were placed into the asynchronous SRAM area.

The *Clock Generation Unit*, which is controlled by an external crystal oscillator, produces a clock signal f_{OSC} at 20MHz. The CPU clock runs at 150MHz, and the system clock at 75MHz.

2.4.4. The C167

The Infineon C167 (more precisely, the C167CR) 16-Bit CMOS Single-Chip Microcontroller has been used in the Challenge by TuBound, via the tool Calc_wcet_167 (see Sec. 3.9). It is a single-issue, in-order architecture with a jump cache. The C16x family of microcontrollers targets real-time embedded control applications and is optimized for high instruction throughput and low response time to external interrupts. It combines features of both RISC and CISC processors. Separate buses connect the program memory, internal RAM, (external) peripherals and on-chip resources. The CPU is clocked at 25/33 MHz allowing a 80/60 ns minimum instruction cycle time. For more details about the C167 microcontroller refer to the manual [21].

The core of the CPU consists of a for 4-stage instruction pipeline, a 16-bit ALU, dedicated SFRs, separate multiply, divide, bit-mask generator and barrel shifter units. Because of optimized hardware, most instructions can be executed in one machine cycle. Instructions requiring more than one cycle have been optimized. Branching, for example, requires only one additional cycle when a branch is taken. The pipeline is extended by a 'Jump Cache' that optimizes conditional jumps performed repeatedly in loops: most branches taken in loops require no additional cycles.

The memory space of the C167 is a Von Neumann architecture, code memory, data memory, registers and IO ports are organized in the same 16MB linear address space. Memory can be accessed byte-wise

[6] `http://www.nexus5001.org/`

or word-wise. Particular portions can be addressed bit-wise, which is supported by special instructions for bit-processing. A 2 KByte 16-bit wider internal RAM provides fast access to registers, user data and system stack.

2.4.5. The JOP Architecture

JOP is a Java processor especially optimized for embedded real-time systems [37]. The primary design target of JOP is time-predictable execution of Java bytecodes, the instruction set of the Java virtual machine (JVM). JOP is designed to enable WCET analysis at the bytecode level. Several Java WCET tools target JOP; WCA [38], the WCET analysis tool that is part of the JOP distribution, was used in the WCET Challenge 2011 (see Sec. 3.10). JOP and WCA are available in open-source under the GNU GPL license.

The JOP pipeline is as simple as the ARM7 pipeline. The main difference is that a translation of bytecodes to a sequence of microcode instructions is performed in hardware. Microcode instructions execute, as in standard RISC pipelines, in a single cycle. Bytecode instructions can execute in several cycles. The timing model for bytecode instructions is automatically derived from the microcode assembler code by WCA.

Bytecode instructions usually execute in constant time. Only for instructions that access main memory the access time has to be modeled. In WCA modeling of a simple SRAM memory is included and also a model of a chip-multiprocessor version of JOP with TDMA based memory arbitration [32].

JOP contains three caches: a stack cache for stack allocated local variables, a method cache for instructions, and an object cache for heap allocated objects. The stack cache has to be large enough to hold the whole stack of a thread. Spill and fill of the stack cache happens only on thread switch. Therefore, a guaranteed hit in the stack cache can be assumed by WCA. The method cache stores whole methods and is loaded on a miss on a method invocation or on a return. WCA includes a static, scope-based persistence analysis of the method cache. The analysis of the object cache [19] is not yet completely integrated into WCA and we assume misses on all object field accesses for the WCET Challenge.

With the method cache JOP is slightly more complex than the ARM7 target. The reference configuration of JOP uses a 4 KB method cache and a 1 KB stack cache. The main memory is 32-bit, 1 MB of SRAM that has a read access time of 2 clock cycles and a write access time of 3 clock cycles.

3. Tools and Experience

An overview of the tools and target processors used in WCC'11 was already given in Table 1. As indicated there, five out of ten tools do flow analysis on the source-code level. This means that their flow-analysis results could in principle be compared in source-code terms. For example, on the source-code level we can talk about iteration bounds for specific loops, which is not possible on the machine-code level because of code optimizations.

On the ARM7, aiT, Bound-T, OTAWA used the gcc ARM 3.4.1 for PapaBench, and the gcc-if07 for debie1. TuBound used the gcc-c16x for the C167. aiT and TimeWeaver used the powerpc-eabi-gcc (Sourcery G++ Lite 2010.09-56) 4.5.1 for the MPC5554. FORTAS used the hightec-tricore-gcc-3.4.5 for the TC1796.

In the following, each tool is briefly described, followed by a report on the experience and results in participating in WCC'11. The descriptions are written by the developers of the tool, edited only for uniform formatting.

3.1. aiT (written by S. Wegener and D. Kästner)

AbsInt's aiT[7] is a timing verification tool. Static WCET analysis is used to compute a safe upper bound of the actual WCET of a task. Its target processors range from simple architectures like the ARM7TDMI to highly complex architectures like the PowerPC 7448.

The main input of aiT is the binary executable, from which the control-flow graph is reconstructed. On this graph, several static analyses take place to compute the execution time of each instruction. A global path analysis is used afterwards to compute the task's overall WCET bound. The results of the analysis are then visualized to the user.

aiT requires no code modification. Its analysis is performed on exactly the same executable that runs on the shipped system. Manual annotations can be used to express known control-flow facts or values of registers and memory cells. aiT supports tight integration with many state-of-the-art development tools, including SCADE Suite[8] and SymTA/S[9]. AbsInt offers qualification support for aiT in DO-178B [35] qualification processes (up to level A).

aiT has been successfully used for timing verification in the avionics, aeronautics and automotive industries (e. g. [39, 30]). A free trial version can be obtained from AbsInt.

3.1.1. Adapting aiT to the proposed common target architectures

aiT already supported the proposed common target architectures. Thus nothing had to be changed to analyze the benchmarks. Nevertheless, both the control-flow reconstruction part of aiT as well as the loop analysis part have been extended to reduce the amount of annotations that must be manually added to perform the analyses of the benchmarks.

3.1.2. Analysis of the debie1 benchmark

Both the MPC5554 version and the ARM7 version have been analyzed.

A WCET bound could be computed for each problem. For the T2 problems concerning the maximal interrupt blocking times, we are assuming for any function containing the interrupt enabling/disabling macro that the entire function is executed with disabled interrupts. This had to be done because the macros DISABLE_INTERRUPT_MASTER and ENABLE_INTERRUPT_MASTER were defined as no-ops and thus not visible in the binary. Only little overestimation is introduced by this simplification since most routines called the macros directly at the beginning and at the end. As an exceptional case, the routine "RecordEvent" enables interrupts, calls "FindMinQualityRecord" and then disables the interrupts again. Here, the WCET contribution of "FindMinQualityRecord" has been subtracted from the WCET contribution of "RecordEvent" to get the execution time of the interrupt-disabled region.

[7] `http://www.absint.com/ait/`
[8] `http://www.esterel-technologies.com/products/scade-suite/`
[9] `http://www.symtavision.com/symtas.html`

Most loop bounds were derived automatically by aiT. For those loops where this was not the case, loop bound annotations have been added. The input constraints as defined in the WCET Tool Challenge's wiki[10] have been used to write annotations for the analyses. All input constraints except one could be transformed to annotations. The one exception is problem 2c for ARM7. Here, the compiler transformed an if-then-else construct into conditionally executable code. Although aiT analyzes conditionally executable code, there is at the moment no possibility to annotate the state of the condition flags.

The annotations provided to aiT can be found in the wiki of the 2011 WCET Tool Challenge. Also all the flow questions were answered. However, the invocation counts are computed only for the worst-case path. Due to this, the answers of problem 6d, question F1 differ between the ARM7 and the MPC5554. On the latter, "SetSensorUnitOff" is not on the critical path and thus the invocation count is zero (instead of eight on the ARM7).

3.1.3. Analysis of the PapaBench benchmark

Only the ARM7 code in RAM version has been analyzed. A bound[11] could be computed for each problem. One problem during the analysis was that the fly-by-wire executable contains debug information which could not be read by aiT or GNU objdump. Fortunately, the benchmark's creator was able to send a file which contained the mapping between source code lines and binary code addresses. With the help of this file, we could also answer the flow question of problem F1b. Another problem were the loops in the software floating point library. This floating point library is not used by any of our commercial customers and no loop bound annotations were available.

aiT was only able to derive some loop bounds, but not all. To derive the remaining bounds by hand/brain would have required more effort than we were willing to invest. Therefore, we simply assumed 99 iterations for the actual division loop.

From the flow questions, only those regarding feasible or unfeasible code have been answered. The rest of the questions concerned the bounds of angle normalisation loops for which aiT did not automatically find loop bounds. We simply annotated them to iterate once. Our annotations can be found in the wiki of the 2011 WCET Tool Challenge.

3.1.4. Comments on the Daimler benchmark

Comparing the results in Table 7 (Sec. 5.3) for the aiT OTAWA-like MPC5554 configuration and OTAWA MPC5554 for the small code snippets `INTERR` and `INIT`, we see a rather high difference of a factor of around 2.

AbsInt found the OTAWA MPC5554 results to be surprisingly low in some cases and assumed that OTAWA underestimates the WCET. Without access to the Daimler code, we used another executable and used our MPC5554 evaluation board to produce a NEXUS trace for the entry point of a given function in the processor configuration supported by OTAWA. This trace shows an execution time of around 451 cycles, while the OTAWA tool only predicts 320 cycles. We therefore assume that there is a difference in the CPU modeling of aiT and OTAWA and the results are not comparable.

[10] http://www.mrtc.mdh.se/projects/WCC/2011/doku.php?id=bench:debie1, as of May 5, 2011

[11] The correctness of this bound depends on the correctness of our loop bound assumptions.

Unfortunately, it was not possible to get actual hardware measurements from Daimler for the entry points we used.

3.1.5. Comments on the WCET Tool Challenge

We think that the WCET Tool Challenge is very useful for the community. The tool vendors can improve their tools because they get valuable input about the strengths and weaknesses of their tools.

3.2. Astrée (written by S. Wegener and D. Kästner)

AbsInt's Astrée (Analyseur statique de logiciels temps-réel embarqués)[12] is a verification tool to prove the absence of runtime errors in embedded C code compliant to the C99 standard. A free trial version of Astrée can be obtained from AbsInt.

Examples of runtime errors which are handled by Astrée include division by zero, out-of-bounds array indexing, and erroneous pointer manipulations and dereferencing (NULL, uninitialized and dangling pointers). Moreover, Astrée can be used to prove that user-defined assertions are not violated. As an experimental feature, Astrée can export loop bound annotations and function pointer target annotations for aiT.

Astrée is no tool for WCET analysis. However, the information it computes can be used to help the WCET analysis. In this benchmark, Astrée has been used to derive flow information (like loop bounds and function pointer targets) from the C source code.

As Astrée targets C code, no adaptations are required to support various target architectures. The PapaBench benchmark has not been analyzed with Astrée.

3.2.1. Analysis of the debie1 benchmark

Astrée is not directly targeted on flow analysis. However, we were interested how well Astrée can be used to retrieve flow information useful for WCET analysis.

An experimental feature has been added to Astrée to produce loop bound annotations and function pointer target annotations for aiT. To count routine invocations, for each routine of interest, an own static integer variable had been added. These variables are incremented by one for each routine invocation. Astrée's value analysis is then used to derive an interval for these variables. Answers were produced for all flow problems.

The following assumptions have been used during the analysis: (1) Only the tasks of interest have been analyzed, but no initialization routines, because the problem specification stated that any task may run between the invocation of the particular task and its initialization tasks. Thus all possible values have been assumed for those variables that were not initialized inside the analyzed tasks. (2) For those variables where some input constraints were given in the problem description, the constraints have been used to narrow down the value range of these variables.

[12] `http://www.absint.com/astree/`

3.2.2. Comments on the WCET Tool Challenge

The analysis of the debie1 benchmark showed that in principle, Astrée can be used to compute the flow information needed for WCET analysis. The participation of Astrée in this Challenge was particularly interesting for us, since it has been used for a purpose it was not designed for. Some work has already been invested to streamline the export of flow information. We will further investigate the use of Astrée for means of flow analysis.

3.3. Bound-T (written by N. Holsti)

Bound-T is a WCET analysis tool that uses static analysis of machine code to compute WCET bounds and (optionally) stack-usage bounds. Starting from the entry point of the specified root subprogram Bound-T constructs the control-flow and call graphs by fetching and decoding binary instructions from the executable file. Bound-T models the integer computations as transfer relations described by Presburger arithmetic formulas and then analyses the transfer relations to identify loop induction variables and loop bounds and to resolve dynamic branches. Some infeasible paths may also be detected. Various simpler analyses such as constant propagation and copy propagation are applied before the powerful but costly Presburger models. Bound-T is focused on microcontrollers with predictable timing. Caches and other very dynamic hardware components are not considered. The WCET is calculated with the Implicit Path Enumeration technique, applied to each subprogram separately. Bound-T is commercially distributed and supported by Tidorum Ltd.

3.3.1. Bound-T's Participation in this Challenge

Of the target processors suggested for the 2011 WCET Challenge, Bound-T supports only the ARM7. My participation was thus limited to the benchmarks available for the ARM7: debie1 and PapaBench. Both benchmarks have been used in earlier Challenges. For the 2011 Challenge the debie1 analysis problems were slightly changed, based on participant feedback from the 2008 Challenge, so I had to update the Bound-T annotations correspondingly. PapaBench was used in the 2006 Challenge, but not in 2008 when the "analysis problem" structure was introduced, so the PapaBench analysis problems were new. However, I could reuse many low-level annotations from the 2006 Challenge.

The particular ARM7 processor suggested for the 2011 Challenge, the NXP LPC2138, has a feature called the Memory Accelerator Module (MAM) that operates like a small cache and prefetch buffer for the flash memory. Bound-T does not model the MAM but assumes constant memory access times, in effect zero wait cycles for all memory accesses.

3.3.2. Problems with Benchmarks

The capabilities of Bound-T have evolved only a little since the 2008 Challenge, so all of the difficulties with debie1 in the 2008 Challenge are still present, for example the inability to find and analyse the WCET of interrupt-disabled code regions, as required by the debie1 analysis problems 5a-T2 and others. Many of the constraints and assumptions in the debie1 analysis problems cannot be expressed as such in the Bound-T annotation language, but must be translated into different kinds of annotations that have the same effect on the analysis. For example, there is no way to assert that a variable does not have a specific value, as required by the debie1 analysis problem 2a. This translation requires study of the benchmark source code and is not always easy.

PapaBench created new problems, some of which were quite different from the debie1 problems. In PapaBench, almost all loops aim to normalize floating-point variables, representing angles, to some basic "unwrapped" range, for example 0 to 360 degrees. The loops do this by repeatedly adding or subtracting a full circle until the basic range is reached. Bound-T does not attempt to find bounds for loops where termination depends on floating-point conditions, so I had to find loop bounds manually. This meant finding bounds on the value of the angle variable on entry to the normalisation loops. This was tolerably easy for some cases, but too hard for me in other cases, for which I made a guess at the loop bounds.

The PapaBench analysis problem A5 asks for the WCET of a part of a C function. Bound-T does not have a general "point-to-point" analysis capability, but in this case I was lucky: the interesting part is the tail end of the function, so I could tell Bound-T to analyse the "function" starting at the first instruction in the interesting part, as if this instruction were the entry point of a function, and go on in the normal way to the return instruction.

3.3.3. Conclusions for Bound-T

As Bound-T had been applied to both benchmarks in earlier Challenges, the 2011 Challenge did not reveal any new problems or inspire new ideas for improving Bound-T. However, it was a useful reminder about the problems with translating constraints from the conceptual, application-oriented level to concrete, code-oriented annotations. This is a gap that should be filled, but filling it may need new ideas for representing really high-level constraints in WCET analysis.

3.4. FORTAS (written by S. Bünte, M. Zolda, and R. Kirner)

FORTAS (the *FORmal Timing Analysis Suite*) derives WCET estimates of software tasks running on embedded real-time systems. FORTAS is based on a hybrid approach that combines execution time measurements with static program analysis techniques and follows the general principles of *measurement-based timing analysis* (MTBTA) [41].

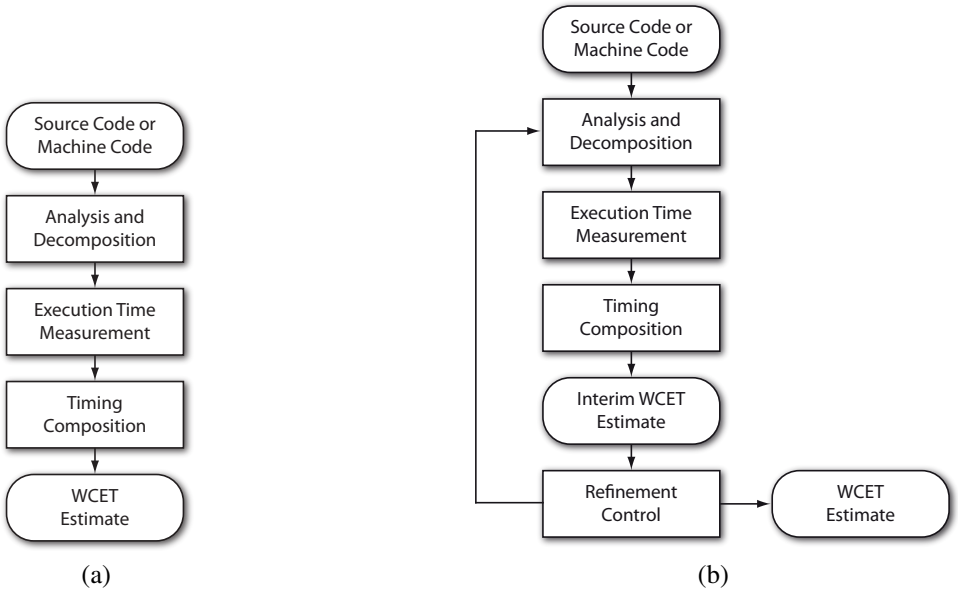

Figure 1: Traditional MBTA work-flow (a) versus the FORTAS work-flow (b).

A central new architectural feature of FORTAS is feedback, as illustrated by Figure 1. The classic MBTA approach as shown in Figure 1 (a) consists of three phases: analysis and decomposition, execution time measurement, and timing composition [41]. In MBTA the temporal behavior of the target system under investigation, in particular of the hardware part, is unknown prior to the first run of the analysis. However, each measurement that we perform on the target system reveals new information. This allows us to initially build a coarse timing model that we refine progressively, as more information becomes available. The benefit of this approach is that we can quickly obtain a rough estimate of the WCET. At a later time, after analysis has passed a few refinement steps, we can check back to retrieve a more precise estimate. Unlike many other approaches, such an analysis does not devise one ultimate WCET estimate—it rather produces an ongoing sequence of progressively more precise estimates. In practice, the estimate converges quickly to a sufficiently stable value, such that the analysis can be finished.

In the FORTAS approach we extend the traditional MBTA approach by an iterative timing model refinement loop, as shown in Figure 1 (b). The refinement is subject to *Refinement Control* that works towards the specific needs of the user. For example, after inspecting an interim timing model, the user can control whether the analysis should focus on minimizing pessimism, i.e., to limit the potential WCET overestimation due to abstraction, or on minimizing optimism, i.e., to limit the potential WCET underestimation due to insufficient coverage of temporal behavior. Refinement Control then adjusts the dedicated input data generation techniques [4, 3] and timing composition [43] methods to be used in subsequent refinement iterations.

3.4.1. Porting the Benchmarks

We ported PapaBench to the TC1796 (described in Sec. 2.4.3) and analyzed problems `A1`, `A2A`, `F1a`, `F1b`, and `F2`.

- We removed the scheduler and analyzed each target function of the respective benchmark problem in isolation. Code that is not needed for a problem is omitted. Analyzing the whole source code in its original version is not feasible with our input data generation technique.
- We annotated trigonometrical functions from our TC1796 `math.h` with *assume statements* of the model checker CBMC to restrict the domain of function arguments. We did this to partly re-incorporate context information that had been lost by removing the scheduler.
- We added start-up code that initializes the processor. The code manipulates TriCore-specific registers to set the CPU clock to a frequency of 150MHz.
- We changed the benchmark to emulate certain accesses to memory registers by global variables. For example, the call of the macro `SpiIsSelected()` was substituted by a read access to a global variable `spi_is_selected`.
- We expanded the preprocessor macros and moved some C expressions and statements to dedicated source code lines, in order to get a canonical version that is interpreted consistently among all FORTAS tools. For the same reason we made short-cut evaluation in decisions and conditional expressions explicit, i.e., we translated such conditionals to semantically equivalent cascades of `if`-statements.
- We converted loops, so that iteration bounds are easily found by CBMC.

- We removed `static` and `inline` declarations without changing the program semantics. Also, we substituted `typedef` directives with equivalent types that do not incorporate any `typedef`. The reason for this modification is that these features are not supported by our prototype.

The transformed versions of PapaBench can be downloaded from our website[13].

3.4.2. Analysis

With our prototype implementation we can analyze ANSI-C code. We use HighTec's GCC version 3.4.5 to compile the source code for our target processor, the TriCore 1796. We then executed the program on the target platform and captured cycle-accurately time-stamped execution traces using a *Lauterbach LA-7690 PowerTrace* device that is connected to the target system via the processor's *On-Chip Debug Support* (OCDS) *Level 2* debugging interface.

Internally, our tools work on a CFG representation of the software under analysis. Loop bounds and other flow facts are currently provided by the user. In the current setting we turned optimization off when compiling the benchmark sources. This is needed in the current implementation stage of the prototype implementation. But this will not be needed in the future, as we have recently shown within the research project SECCO[14] that we can achieve quite high optimization while still maintaining preservation of structural code coverage criteria.

We automatically generate suitable input data using a model-checking based method [15, 16] that has been implemented as the FSHELL[15] tool. FSHELL itself is based on the C Bounded Model Checker (CBMC) version 3.8 [5]. The input to FSHELL is a test suite specification, expressed in the *FShell Query Language* (FQL) [17].

3.4.3. Challenges and Lessons Learned

We encountered several limitations of our analysis tool, most of which are due to the nature of our prototypical implementation: we had to change the benchmarks manually (see above) in order to make them work with FORTAS, which took far more time than we expected. However, those issues can be resolved given sufficient engineering resources to resolve those prototypical deficiencies.

However, some limitations are specific to our analysis approach: the reason why we cannot analyze problems A2b and A3-A6 is due to limitations of our input data generation techniques. Our version of CBMC utilizes an SMT solver that cannot find models for the respective problems efficiently. We suspect the combination of floating point variables and multiplication operations to be the source of the problem. This seems to point at a need for complementary generation methods for input data.

3.4.4. Comments on the WCET Tool Challenge

First, our research benefits from the extended pool of benchmarks. Second, some of the encountered limitations will drive us both in terms of tool engineering and in addressing the problem of input data generation in our future research.

[13] `http://www.fortastic.net/benchmarks_wcc_2011.zip`
[14] `http://pan.vmars.tuwien.ac.at/secco/`
[15] `http://code.forsyte.de/fshell`

Unfortunately, our prototype implementation is not compliant to any of the target processors that are officially supported by the Challenge. Also, we did not have the resources available to add another target system to our tool. Retargeting an MBTA tool to a new target platform requires considerably less effort than in the case of a static WCET analysis tool, but still needs some effort to set up the tool chain. Results from a comparison to other analysis techniques would probably emphasize and reveal both strengths and limitations of our analysis approach that we are currently not aware of.

3.5. METAMOC (written by M. C. Olesen)

METAMOC [6] analyses WCET problems by converting the CFG of a program into a timed automata model, which is combined with models of the execution platform (pipeline, caches). The combined model is then model checked using the UPPAAL model checker, asking for the maximal value the cycle counter can attain, which is then the WCET estimate. No automated flow analysis is implemented, so all flow facts and loop bounds have to be manually annotated, either in the C source code, or by modifying the resulting model. Non-determinism is used to explore all branches, and can therefore also be used in the annotations, if there are uncertainties. Of course, the less precise the annotations the more possibilities the model checker has to explore, and too little precision results in the model checker running out of memory.

3.5.1. Experience

The WCET Challenge was the first time we applied our tool to a real-world benchmark. As such, we were not able to solve many of the problems, but our tool gained many improvements, and it is much clearer what directions to work in to improve the tool. The main problem we encountered in applying METAMOC was getting annotations of a good enough quality. Particularly the floating point routines compiled in by GCC are of crucial importance: they are called very frequently so therefore the annotations need to be of high quality (to limit the possible paths through the function), but on the other hand the routines are highly optimized so therefore hard to analyse.

3.5.2. Comments on the WCET Tool Challenge

As discussed with the steering committee, it might make sense to have two phases to the Challenge: determining loop bounds by all the tools capable of doing so, whereafter these results are shared, so that in the second phase (the WCET analysis phase) all tools use the same loop bounds. This would split the Challenge much more into the two subproblems, which is an advantage for tools focusing on one of the two subproblems.

3.6. OTAWA (written by A. Bonenfant, H. Cassé, M. de Michiel, and C. Rochange)

OTAWA [2] is a library dedicated to the development of WCET analyzers. It includes a range of facilities such as:

- loaders
  - to load the binary code to be analyzed. Several ISAs are supported: PowerPC, ARM, TriCore, Sparc, HCS12. New binary loaders can be generated with the help of our GLISS tool [36].

  - to load a description of the flow facts (loop bounds, targets of indirect branches, imposed branch directions). To handle complex flow facts, the description can be supplemented with a set of hand-written constraints to be added to the ILP formulation used to compute the WCET (IPET [27]).
  - to load a description of the target hardware (processor, memory hierarchy, memory map, etc.). Only generic architectures can be described that way: specific targets need the user to write specific analyses where needed.

- annotation facilities (called *properties*) that make it possible to annotate any object (instruction, basic block, etc.) with any kind of value. They are used to store the results of the successive analyses.

- code processors that use already-computed annotations and produce new ones. Built-in code processors include a CFG builder, a CFG virtualizer, loop dominance analyzers, support for abstract interpretation, hardware analyzers (pipeline, caches, branch predictor) and a WCET computer based on the IPET method (with the help of the `lp_solve` tool).

The library comes with a set of built-in tools that check for absolutely-required flow facts, dump the CFG in various formats (e. g. dot), compute a WCET following an input script that describes the specific analyses to be applied, etc. These tools are also available in an Eclipse plugin.

OTAWA is open-source software available under the LGPL licence[16].

3.6.1. Problems and solutions

Both the recommended targets, namely the PowerPC MPC5554 and the ARM LPC2138 have been modeled in OTAWA. However, we discovered that the PowerPC version of the debie1 benchmark includes VLE instructions which are not supported by OTAWA so far. Then we decided to focus on the ARM target.

The problems we have encountered are all related to flow facts. Some are inherent to the code of the benchmarks, others come from the questions we had to answer.

General difficulties. To compute loop bounds automatically, we use the oRange [29] companion tool developed in our group. It works on the source code. Unfortunately, oRange was not able to determine all the bounds:

- for some of the problems, the source code of some functions was missing (e. g. debie1 5a, PapaBench F1a);

- the increment of some of the loops (e. g. in debie1 6b) could not be computed.

In such cases, we determined the bounds by hand, with success for most of them. This is a fastidious and error-prone work. For functions (e. g. `memcpy`) from the `glibc`, we considered the source codes found on the GNU web site.

[16] `www.otawa.fr`

For some functions, we have found several possible sets of loop bounds (e. g. the bounds for $loop1$ and $loop2$ are either x and y, or x' and y' respectively). This cannot be directly specified to OTAWA. In such cases, we have added appropriate constraints on the sum of iterations of both loops.

Difficulties related to the Challenge questions. Several questions required considering specific switch cases. Our simple flow facts description language does not support this kind of annotations. Then we added hand-written constraints to the integer linear program used to compute the WCET. Said like this, it seems fastidious but in practice it is quite easy thanks to an efficient CFG displayer that shows various information like basic block numbers, branch directions, related source code lines, etc.

Problem 3b for debie1 raised the difficulty mentioned above since it implied that one of two identical loops ends after one iteration instead of processing to the end value. We had to hand-write additional constraints.

3.7. SWEET (written by J. Gustafsson and N. M. Islam)

SWEET (Swedish WCET Analysis Tool) is a WCET analysis research tool from MDH. It has a standard modular tool architecture, similar to other WCET analysis tools, consisting of (1) a flow analysis, where bounds on the number of times different program parts can be executed are derived, (2) a low-level analysis, where bounds on the time it might take to execute basic blocks are derived, and (3) a WCET estimate calculation, where the costliest program execution path is found using information from the first two phases.

SWEET analyzes the intermediate format ALF [11]. ALF has been designed to represent code on source-, intermediate- and binary level through relatively direct translations. Given a code format, SWEET can perform a WCET analysis for it if there is a translator into ALF. Currently, two translators exist: a translator from C to ALF from TU Vienna, and an experimental translator from PowerPC binaries. The first translator enables SWEET to perform source-level WCET analysis. This translator has been used in the WCET Challenge.

The current focus of SWEET is on automatic program flow analysis, where constraints on the program flow are detected. SWEET's program flow analysis is called *abstract execution* [12]. This is a form of symbolic execution, which is based on abstract interpretation. It can be seen as a very context-sensitive value analysis. Abstract execution executes the program in the abstract domain, with abstract values for the program variables, and abstract versions of the operators in the language. In the current implementation of SWEET the abstract domain is the domain of intervals. SWEET's program flow analysis is *input-sensitive*, meaning that it can take restrictions on program input values into account.

SWEET derives program flow constraints on so-called execution counters. These are virtual variables that are associated with program points. They are initialized to zero, and then incremented by one each time the program point is traversed in the execution. The abstract execution will derive program flow constraints in the form of arithmetic constraints on the values of the execution counters: for instance, if the analysis encloses the possible values of the execution counter for a loop header within an interval, then the end-points of the interval provides lower and upper bounds for the number of iterations of that loop. SWEET's abstract execution can also find considerably more complex constraints than simple loop bounds, for instance lower and upper (nested) loop bounds, infeasible nodes and edges, upper node and edge execution bounds, infeasible pairs of nodes, and longer infeasible paths [12].

However, the PapaBench program flow problems mostly consider simple loop bounds only.

SWEET can handle ANSI-C programs including pointers, and unstructured code. It has a large set of options for fine-tuning the analysis. It has an annotation language, where the user can provide additional information that the automatic analysis for some reason fails to derive.

Since SWEET does not support the target processors in the WCET Challenge 2011, we have only performed source-level program flow analysis. We have restricted the analysis to the PapaBench code; debie1 was excluded due to lack of time, and the Daimler code was excluded since we anticipated problems for the students performing the analysis on the Daimler code to use our tool. In particular, the C to ALF translator is hard to install due to its many dependencies to different software packages. We also know by experience that production source code for embedded systems can pose many problems for source-level analysis tools, since such code often stretches the C standard [28].

3.7.1. The flow analysis problems

We were able to obtain answers to all six PapaBench flow analysis problems. In particular, SWEET managed to find bounds also for the floating-point controlled loops in problems A1, and A2a. Due to the input-sensitivity of SWEET's flow analysis, we were able to derive bounds for these that are conditional on certain input values. These bounds are more precise than bounds that have to be valid for all possible input value combinations. The conditions, in the form of input ranges for certain input variables, were found by a search running SWEET with different input ranges for these variables. Interestingly, for problem A2a our analysis also found a possible division by zero if the input variable `estimator_hspeed_mod` is zero. If this variable indeed can assume this value, then there is a potential bug in PapaBench.

For some problems we had to tweak the code, or take some other measures, to make the analysis go through. For problem A3, we had to remove the "inline" keyword at three places since our C to ALF translator did not accept this use of the keyword. The code for problem F1b contains an infinite loop: we had to patch the code to make this loop terminate to perform the analysis.

At some places in the PapaBench code, absolute addresses are referenced. Such references are problematic when analyzing unlinked source code, since potentially any program variable can be allocated to that address when the code is linked. Thus a safe analysis must assume that the absolute address can be aliased with any program variable, and this is indeed what SWEET assumes by default. However, this will typically lead to a very imprecise, more or less useless analysis. To remedy this, we equipped SWEET with a mode where it assumes that all absolute addresses are distinct from all unallocated program variables. This is often a reasonable assumption, since absolute addresses typically are used to access I/O ports and similar which are distinct from data memory. In addition, the analysis assumes that absolute addresses always hold the abstract TOP value (no information about its possible value), since the value of input ports and similar can be altered from outside the program. In all but very unusual situations, an analysis resting on these assumptions should be safe.

3.7.2. Conclusions, and lessons learned

SWEET was able to solve all six program flow analysis problems posed for PapaBench automatically. Notably, these problems include loops that are controlled by floating-point variables. We had to tweak the source code at some places to make all analyses go through: however, these tweaks were

necessitated by current limitations in translator and analysis tool that are not of fundamental nature, and fixing them should be a mere matter of engineering.

We got much valuable feedback during the process of analyzing PapaBench, which enabled us to improve our tool chain. The introduction of a mode to handle references to absolute addresses has already been mentioned. In addition a number of bugs were revealed in both the C to ALF translator, and SWEET.

Acknowledgments We are grateful for the support given by Andreas Ermedahl, Linus Källberg, and Björn Lisper.

3.8. TimeWeaver (written by S. Wegener and D. Kästner)

AbsInt's TimeWeaver is a measurement-based timing estimation tool. It can be used for any processor with NEXUS-like tracing facilities[17], i.e. with hardware support for non-invasive tracing mechanisms. TimeWeaver's main focus is not timing verification but exploring the worst-case timing behavior on actual hardware and identifying hot-spots for program optimizations. A free trial version of the TimeWeaver prototype can be obtained from AbsInt.

The main design goal for TimeWeaver was simplicity. After specifying the set of input traces and the analysis starting point, TimeWeaver is able to compute a WCET estimate in a fully automatic way. All the needed information is taken from the measurements. At the current point of time, no additional knowledge can be added by annotations. If for example a loop has at most five iterations in the traces, but the assumption is that the particular loop has a bound of ten iterations, the analysis is only able to use the bound of five. Unfortunately, this hampers the comparability of TimeWeaver with other WCET tools, but on the other hand, it eases the use of TimeWeaver.

To compute a WCET estimate, an ILP is constructed from the traces which represents the dynamic control-flow graph as observed by the measurements. Loop bounds and time stamps are also extracted from the traces.

3.8.1. Adapting TimeWeaver to the proposed common target architectures

As TimeWeaver works on NEXUS traces, only the MPC5554 was considered as target. For this processor, a prototype already existed. This prototype has been extended to handle incomplete traces. Moreover, the handling of routines with multiple exits has been improved.

3.8.2. Analysis of the debie1 benchmark

The debie1 benchmark was the only one which was analyzed with TimeWeaver because it was the only one available for the MPC5554. Since TimeWeaver is a measurement-based tool, the quality of the results depends heavily on the quality of the input traces. Unfortunately, the measurement solution used to get the traces showed some unforeseen problems (see next section). No comparable results were therefore computed by TimeWeaver.

[17] `http://www.nexus5001.org/`

3.8.3. Trace generation problems

The first problem was the lack of automation support of the rather old tracing equipment available at AbsInt. Producing several thousand traces for each task invocation manually one by one would have been a huge effort and was not considered as a practical option. Instead, we tried to trace the harness part as a whole.

This approach uncovered two other problems. First, the distinction between the various subquestions was not possible with the large traces because the NEXUS traces contain only instruction addresses and timestamps. Thus, only estimates for the entire tasks could be computed, without taking the input constraints into account. Second, the trace buffer of the used measurement equipment is of only limited size. Thus sometimes the traces ended prematurely and no full path coverage was achieved.

3.8.4. Comments on the WCET Tool Challenge

For the next incarnations of the Challenge, we believe that having a standard set of measurements would be a tremendous advantage. Then, all measurement-based tools could use the same input, thus enabling more room for comparison. Moreover, having traces of the worst-case paths would also ease the comparison between the actual WCET and the computed estimates. Last but not least, this would prevent the participants from suffering from the same problems we had.

Overall, we think that the Challenge is a good source of inspiration, as we have found some things that could be improved in our prototype. We hope that next time we can produce all the needed traces without problems, thus allowing more room for comparison with other tools.

3.9. TuBound (written by Adrian Prantl and Jakob Zwirchmayr)

TuBound is a research prototype WCET analysis and program development tool-chain [33] from Vienna University of Technology, built on top of libraries, frameworks and tools for program analysis and transformation. Flow information is acquired and annotated (either supplied by the user or inferred by an analyzer or a software model checker) at source code level. TuBound's loop bound analysis component was recently extended by SMT reasoning to rewrite multi-path loops into single-path ones. Additionally, certain classes of single-path loops are translated into a set of recurrence relations over program variables, which are then solved by a pattern-based recurrence solving algorithm. The extension is denoted r-TuBound and described in more detail in [26, 25].

The gathered flow information is conjointly transformed within the development tool chain. The transformed annotations are further used by the WCET analyzer to calculate the WCET.

TuBound combines a C/C++ source-to-source transformer (the ROSE compiler framework), static analysis libraries (SATIrE, TERMITE), used to implement a forward-directed data flow interval analysis, a points-to analysis and a loop bound analysis, a WCET-aware C compiler (based on GNU C compiler 2.7.2.1 ported to the Infineon C167 architecture with added WCET analysis functionality), and a static WCET analysis tool. The WCET analysis tool currently integrated into the TuBound tool-chain is Calc_wcet_167, a static WCET analysis tool that supports the Infineon C167 as target processor. Further details about TuBound can be found in [34, 33].

3.9.1. TuBound—Target Architecture

Although TuBound is conceived from ground up to be modular and to support multiple WCET analysis back ends, TuBound currently only supports the Infineon C167 architecture, described in Sec. 2.4.4). Because the development team of TuBound is in transition there has not been advancement in interfacing other WCET back-ends and therefore supporting additional target architectures. Implementation and evaluation of the r-TuBound extension and the WCET Challenge showed that the focus of development needs to include support for different architectures, such that results can be compared to the results of other tools.

3.9.2. TuBound Problems with Benchmarks

General In some cases it was not possible to annotate the input constraints because there is no support for them in TuBound. For example, TuBound supports neither path annotations specifying "the first run" (or in general the xth run), nor constraints that specify that "function f is executed once before g." Additionally, the interval analysis does not support arbitrary user supplied value annotations. Some of the input constraints can nevertheless be annotated manually. For the cases where the input constraints could not be annotated fully, we report the worst-case result. Therefore, for example, when the WCET of "the first run" of a function is asked for, we calculate the WCET of the function and use it as result. If there are constrained inputs that we cannot model, we again compute the (general) WCET of this function and report it as an over-approximation of the WCET of the run in question.

Another difficulty stems from the supplied assembler code: we cannot perform WCET calculation for the assembler code, because we do not support the target architecture. Therefore we could not, for example, find out the WCET of interrupt routine `__vector_10`.

Another feature TuBound is still missing is floating point support: interval analysis does not consider float values; those are used, for example, in parts of the PapaBench inputs.

Tool Challenge We expected that our r-TuBound extension would improve the quality of our results, as it allows us to find bounds for loops that were previously unbounded by TuBound. Nevertheless the expectations were not met, partly because TuBound's loop analysis already performed well enough on the loops found in the debie1 benchmarks and partly because the benchmarks from the PapaBench suite were not as loop-centric as we expected. The upper loop bound problems in PapaBench all involved floats, which we do not handle, yet, even though basically the loops could be bound by our loop analyzers if there was value information available.

We gained valuable experience from the benchmarking part of the Challenge that took place at Daimler. Even though we do not support the target architecture of this application, we hoped to infer at least flow information from these sources. The outside evaluation of the tool on industry benchmarks at Daimler showed, though, that we need to work on a shippable, binary version of TuBound (we have some licensing restrictions on our version of PAG and EDG, both used in TuBound), because the compilation and editing effort for a running version of TuBound is quite high (e. g. the ROSE compiler infrastructure, the SATIrE framework and the TERMITE library). Additionally, there are portability issues in TuBound that need to be addressed (e. g. hard-coded paths). Therefore some effort should go into creating a central configuration file to ease the configuration of TuBound. This might not be necessary for a research prototype that is always evaluated inside the institute, but it is

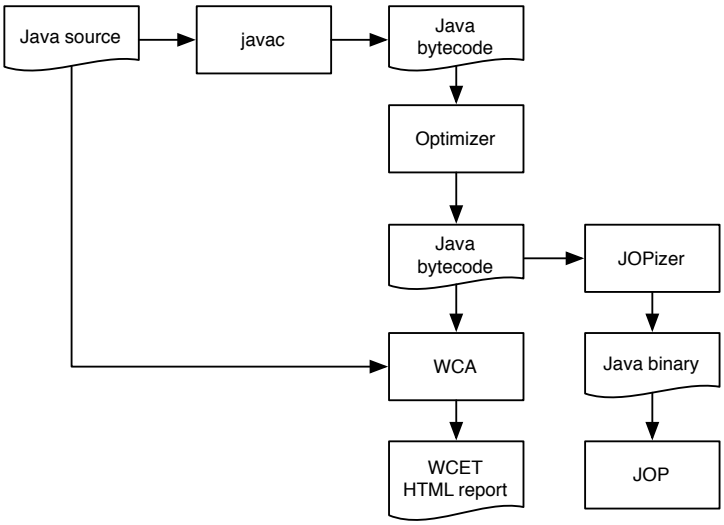

Figure 2: Tools and compilation, optimization, analysis, and build flow for JOP.

necessary for outside evaluation of our tool where sources are not available.

Acknowledgments Thanks to Raimund Kirner for assistance with the Calc_wcet_167 tool, and to Jens Knoop and Dietmar Schreiner for the discussion of and support for legal issues.

3.10. WCA (written by B. Huber, W. Puffitsch and M. Schoeberl)

The WCA tool from Vienna University of Technology and DTU is a static WCET analysis tool for processors executing Java bytecode, currently only supporting JOP [38]. The input to the analysis tool are Java class files, along with information on the processor configuration. The latter consists of hardware parameters, such as cache sizes and memory access timings, and of the microcode assembler code for each bytecode.

Figure 2 gives an overview of the tools and the build and analysis flow. Java source, with optional loop bound annotations, is compiled with a standard Java compiler to Java bytecode. The optional optimizer uses bytecode as input and produces bytecode. It is planned to integrate the optimization process with WCA to perform WCET based optimizations. The bytecode is the input for the WCA tool that produces reports in HTML. WCA also reads the Java source to extract annotations. The bytecode is also the input for the tool JOPizer to generate a linked executable, which can be downloaded to JOP. Details of generating the processor JOP and automatically deriving the timing information are omitted from the figure.

For the high-level path analysis, bytecode has several advantages compared to machine code. Most type information is still present in bytecode, even automated decompilation is feasible. In particular, it is easy to automatically obtain control flow graphs from bytecode. The possible targets for indirect branches (switch) are specified in the class file. Instead of indirect function calls, bytecode solely relies on dynamic method dispatch.

Determining the methods possibly executed due to a virtual invocation amounts to determining the dynamic type of the receiving object. To this end, WCA includes a data flow analysis (DFA) to determine precise dynamic types of objects, which is also used to prune the call graph. Additionally,

the DFA computes bounds on the range of values. This information is used for a simple loop bound analysis, which makes it unnecessary to manually analyze and annotate many loops, and to obtain the size of arrays for analyzing allocation rates. Manual loop bounds may be provided at the source code level. The annotation language supports bounds relative to outer loops and symbolic expressions. In particular, it is possible to refer to Java constants in loop bound expressions, which reduces the maintenance burden considerably.

The pipeline analysis for JOP is relatively straightforward. One distinguishing feature of WCA is that it derives a symbolic formula for the worst-case execution time of bytecode instructions automatically. To this end, the microcode sequence executed for a bytecode is inspected. The analysis composes a formula which takes explicitly hidden memory latencies and method cache accesses into account.

WCA also includes a static analysis for JOP's method cache. It implements a scope-based persistence analysis for the N-block method cache with FIFO replacement. This analysis inspects program fragments, and tries to prove that, within one fragment, at most N cache blocks are accessed. If this is indeed the case, method cache costs only need to be accounted for once for a method accessed within the fragment. This is encoded in the IPET formulation, using a standard technique adding cache miss and cache hit variables.

Although WCA is a command line tool, it produces annotated, colored listings of Java code, which can be used to inspect the worst-case path. As we maintain relatively precise bytecode to source code line mappings, this can be done on the Java source code, which is definitely more pleasant than inspecting low level code.

The combination of WCA and JOP is a little bit different from the other tools participating in the Challenge as we support Java instead of C. Therefore, we had to port the benchmarks to Java. Furthermore, the different languages and the different execution platform make is problematic to compare WCA with the other tools.

3.10.1. Porting the Benchmarks to Java

While we could reuse the Java port of Papabench from Michal Malohlava [24], the debie1 benchmark was ported by ourselves. Unfortunately, the port of Papabench is incomplete. As we did not want to deviate too far from the publicly available version of the benchmark, we fixed only a few minor issues, but left the general implementation of the benchmark unchanged. One notable change in the implementation was the use of scoped memory to enable dynamic memory allocation while avoiding garbage collection. Due to the incompleteness of the benchmark, we were only able to answer a few questions posed by the Challenge. In order to provide a more complete picture, we include the analyzed and observed WCETs of the benchmark's tasks in Table 4.

debie1 was ported as far as necessary to properly execute the test cases provided in the harness. However, some functionality was omitted as it would not have been possible to test the respective code properly.

During porting, we encountered a few advantages and disadvantages of Java. In C, structs are laid out flat in memory and can be accessed byte for byte through a pointer. In Java, accessing an object byte for byte requires manual mapping of byte indices to fields, which is considerably more expensive. A related issue are accesses to multidimensional arrays. While in C it is possible to use a unidimensional

index to access elements in such an array, this is not possible in Java. For accesses to a multidimensional array in Java, it is necessary to compute the correct index for each dimension, which requires a division and remainder operations. If strength reduction is not possible, this introduces severe overheads.

Enumerations in C are extremely light-weight, but are full-blown objects in Java. On a few occasions, we preferred integer constants over enumerations to avoid the respective overheads.

Java has a clear concept for modularization. While it is still possible to write poorly modularized code, the object orientation of Java serves as gentle reminder to programmers. Also, being able to control the visibility of fields encourages clean interfaces. Some of the arguments above are against Java in real-time systems due to the considerable overhead inherited by an object-oriented language. However, it should be noted that Java with its strong typing and runtime checks is a safer language than C and therefore, in the opinion of the authors, an interesting choice for safety-critical applications.

3.10.2. Problems and Insights

debie1 The main problem in the analysis of debie1 (in particular Problem 1 and Problem 3) is that methods tend to be very long. We usually assume that in safety-critical code, methods are kept short, as recommended by safety-critical programming guidelines (e. g., [18]). In our "literal" port of the debie1 benchmark to Java, there are many very long methods along with very large switch statements. First, the method cache of JOP can be rather inefficient for very long methods. Secondly, our cache analysis uses rather coarse-grained scopes (methods only) for persistence analysis, and therefore delivers poor results for Problem 1 and Problem 3. From the analysis point of view, considering subgraphs as persistency scopes would considerably improve the analysis. Another attractive option is to automatically refactor large methods into smaller ones. A related problem is the use of switch statements to implement what usually would be realized using dynamic dispatch in Java. This leads to very large methods, which severely impact the method cache performance, even in the average case. Again, refactoring to more idiomatic code (Command Pattern [7]) would resolve this problem.

We replaced all the preprocessor-based configuration in the original debie1 code by methods of a Java interface, which abstracts the actual hardware. In order to eliminate the resulting efficiency penalty, it is necessary to have an optimizer to remove this indirection once the configuration of the hardware platform is fixed. An optimizer for Java bytecode is currently under development, which includes method inlining. As this optimizer is still under development, the execution time for the interrupt handling routines currently is very high.

On the positive side, we used the chance to improve our annotation language, which now supports arbitrary expressions involving Java constants. For example, the annotation for the checksum calculation is

```
// @WCA loop <= union(CHECK_SIZE, 1 + CODE_MEMORY_END
// - MAX_CHECKSUM_COUNT * CHECK_SIZE)
```

where `CHECK_SIZE`, etc. are Java constants defined in the code.

The results for debie1 are given in Table 3. To show the effectiveness of the method cache analysis we also show analysis results with the assumption of all misses in the method cache and all hits in the

Problem	all-miss	all-hit	WCET	Measured
(1)	19111	12719	17717	6977
(2a-2c)	9960	7385	9104	6601
(3a-3c)	158549	120561	132353	67666
(4a-4d)	32150	24419	26863	24652
(5a-5b)	1661×10^3	1371×10^3	1382×10^3	1289×10^3

Table 3: Analysis results for jDebie problems (in clock cycles).

Task	all-miss	all-hit	WCET	Measured
AltitudeControl	33078	27978	29054	23667
ClimbControl	139987	120938	126515	105926
RadioControl	69216	60198	64266	2444
Stabilization	168261	150349	156974	131910
LinkFBWSend	21 (empty)			0
Reporting	21 (empty)			0
Navigation	cyclic CFG			3057905
CheckMega128Values	9710	8618	9710	9417
SendDataToAutopilot	11692	10104	11574	393
TestPPM	4633	3341	4629	610
CheckFailsafe	cyclic CFG			515

Table 4: Analysis results for jPapabench tasks (in clock cycles).

method cache (in the second and third columns). The WCET analysis result must lie between these extremes.

For Problem 6, we did not find meaningful flow constraints, and thus failed to determine a reasonable WCET bound. We did not work on the flow analysis subproblems, lacking support for artificial flow constraints, and only analyzed the worst-case path for each problem. Although we prefer to minimize the use of manual annotations, after working on the debie1 problem set we believe an interactive tool to explore different paths would be a valuable addition to WCA.

Papabench Papabench was relatively straightforward to analyze, even though our value analysis could not cope with the multi-threaded code. In fact, only two (symbolic) loop bounds had to be annotated in the application code. However, the use of floating-point operations proved problematic. On the one hand several loops with non-obvious bounds had to be annotated in the software implementations of these operations, on the other hand the resulting execution times were less than satisfying, both in analysis and measurements. Although we were able to correctly bound the execution times for the floating-point operations, we do not think that such code is suitable for embedded applications. Figure 4 shows the analysis results and execution time measurements.

3.10.3. Remarks on the WCET Challenge

We learned a lot, and got many new ideas for improving the WCET tool (especially with respect to interactive exploration of worst-case paths). Though porting and analyzing the benchmarks was a lot

Benchmark	**debie1**		**PapaBench**		**Daimler**
Type of question	Flow	WCET	Flow	WCET	WCET
Number of questions	15	22	6	11	4
aiT	15	22	3	11	4
Astrée	15				
Bound-T	14	18	6	11	
FORTAS				5	
METAMOC					
OTAWA	8	15	5	11	4
SWEET			6		
TimeWeaver		6			
TuBound	15	18	1	10	
WCA		13		11	

Table 5: Number of posed and answered analysis problems in WCC'11.

of work, we think the participation was a success: We managed to analyze two large benchmarks, and at least for Papabench, the results were quite satisfying.

4. Results

The full set of results is too large to be presented here; please refer to the Wiki. Table 5 shows the number of analysis problems for each WCC'11 benchmark, the number of flow-analysis and WCET-analysis questions to be answered, and the number of questions answered by each participating tool. If a tool answers the same question for several target processors, it still counts as only one answer.

For the three tools that analyzed the simple processor target (ARM7), Table 6 lists the specific results. As can be seen, most deviations are less than 50%. However, there are notable exceptions that probably deserve further investigation.

5. The Daimler Experiment (written by E. Ploedereder, F. Krause, S. Gepperth, and W. Fellger)

WCC'11 as described so far had the producers of applicable tools bring their intimate knowledge to bear in processing previously available benchmarks. In the Daimler experiment, students of the University of Stuttgart applied the tools to proprietary industrial software (see Sec. 2.2.3). The students had no prior knowledge of either the tools or the analyzed software. They were remotely supported by the tool providers and had access to Daimler employees knowledgeable about the analyzed system.

5.1. The Tools

The target architecture MPC5553 is supported by few of the tools participating in the WCC'11. The experiment was conducted with AbsInt's aiT and with OTAWA, as these tools are the only ones that support the target architecture. It should be noted that OTAWA only supports the MPC 5554 architecture, which might be a reason for the somewhat surprising divergence in the results obtained by the two tools. As a third tool, TuBound had registered for the experiment but we did not succeed

Benchmark Question	**Estimated clock cycles** aiT	Bound-T	OTAWA
debie1			
1	342	333	332
2a	100	93	139
2b	144	143	139
2c	144	104	139
3a	2664	3580	4101
3b	11079	22206	23829
3c	11664	22762	27 117
4a	2352	2343	522460
4b	613	214	210
4c	196	187	195
4d	199	190	730
5a T1	4154	5223	5329
5a T2	172		42
5b T1	38798	39825	55883
5b T2	180		42
6a T1	22203	22765	
6a T2	98		
6b	23100	23741	
6c	40143	42285	
6d	24184	24254	
6e T1	1101107	372148	
6e T2	158		
PapaBench			
A1	1716	1660	1358
A2a	27785	31699	32735
A2b	31482	37181	38112
A3 T1	3404	3849	1119
A3 T2	8938	10484	9863
A4	4182	5986	5953
A5	5435	5131	4782
A6	12051	17378	17422
F1a	4207	7914	7824
F1b	45	43	40
F2	102	100	102

Table 6: Results for WCET analysis questions for the ARM7. The *estimated clock cycles* refer to the results reported by aiT, Bound-T, and OTAWA.

in its on-site installation.

5.2. Experiences with the Two Tools

The analyzed software contains fault branches trapping in infinite loops. Obviously, this cannot be accommodated in a WCET calculation. The fault branches needed to be filtered out to obtain meaningful results.

With aiT the respective branches and function calls leading to infinite loops could be excluded from the WCET calculation. With OTAWA, unfortunately no approach could be identified to achieve such exclusion. Encountering such an infinite loop sometimes led OTAWA itself into an infinite loop, requiring a forced termination. The entry point `TASK` was a case in point. In order to obtain meaningful comparisons, the analysis problems were reduced to those with entry points that do not reach any infinite loop. Hence, for the entry point `TASK` no results are reported.

Apart from this, OTAWA frequently terminated with a segmentation fault when analyzing the Daimler code. It also terminated the Eclipse IDE if the plugin was used. Despite best efforts from both Daimler and OTAWA supporters, these problems could not be resolved in time. A suspected cause might be related to what OTAWA calls "unresolved controls," potential branch instructions in the code that cannot be automatically resolved. They occurred very frequently in the Daimler code, and we suspect that they might have taken a "wrong" choice from the available substitutes. Surprisingly, the analysis of unrelated entry points kept crashing after such a failure was encountered until OTAWA's configuration file was cleaned by hand. These failures further reduced the entry points for which comparable results could be obtained from the two tools.

Absint's aiT was pretty straightforward to use and did not cause any major problems that could not be dealt with quickly; in particular, it could deal with almost all conditional branches without further interaction. we checked the resulting call graphs for overestimation of loop bounds - which were mostly automatically computed - but they were all reasonable.

OTAWA itself does not compute loop boundaries, so they needed to be set statically for every loop. It should be noted that there is an external tool called "oRange" for this job which we did not get to experiment with because of the general stability issues.

5.3. Results

The comparative results consist of three data sets, two for aiT and one for OTAWA. These data sets are:

- aiT configured for the real hardware. This configuration yields proper WCET results for the hardware the code is compiled for.
- aiT configured for comparison to OTAWA results. The hardware configuration is changed to assume the same parameters OTAWA uses in its MPC 5554 configuration.
- OTAWA with MPC5554 configuration. These are the results that are comparable with the second data set of the aiT measurements. As OTAWA does not support the exact hardware configuration the code is written for, this configuration is as close as the experiment could get to reality.

OTAWA offers predefined configuration "scripts" with very few options, while aiT presents an almost overwhelming range of settings. For aiT, we made use of the option to initialize the CPU settings from actual CPU status registers for the *real hardware configuration*.

Entry point	aiT		OTAWA
	Compiled hardware configuration	OTAWA-like configuration	
`INTERR`	524	204	113
`INIT`	1055	494	218
`CALC1`	2124	830	722
`CALC2`	16507	6218	7991

Table 7: WCET computation results for the Daimler code experiment.

The loop boundaries used for OTAWA are slightly overestimated compared to aiT, as each loop count can only be set globally, not per call to the containing function. The context sensitivity of loop bound estimation in aiT is particularly noticeable in `CALC2`, the only entry point for which the OTAWA result is higher than the corresponding aiT result.

5.4. Conclusion on the Daimler experiment

In order to arrive at comparable numbers, we reran aiT with a CPU configuration approximating the configuration used by OTAWA to get anywhere near comparable results. While these were in fact significantly closer, OTAWA still tended to give lower numbers than aiT.

In searching for causes of the remaining divergence, we traced the estimates down to the individual basic blocks. Even at this level, the estimates by OTAWA remained consistently lower, which makes it very likely that there are hidden differences in the CPU modeling of the two tools that account for the spread in numbers. The OTAWA support concurred in this being a likely cause.

Unfortunately, no actual hardware or faithful emulator was available to the experiment in order to measure actual performance and compare it to the predictions in order to determine how close the predictions came to reality and whether any numbers were underestimations for the actual hardware. AbsInt had hardware available and undertook a test of this hypothesis. AbsInt reports on the results in Sec. 3.1.4. This report supported our impression that it is very important to ensure a precise match of the detailed hardware description to the actual hardware in arriving at meaningful WCET answers that reflect reality or that allow a comparision of numbers obtained by different tools.

6. Conclusions

One of the goals formulated in the conclusions of the last Challenge, WCC'08 [13], was "to motivate the 2006 participants to rejoin the Challenge, without losing the new 2008 participants." We have adopted this goal, and wanted to provide a setting that would be attractive to as many participants as possible, irrespective of whether or not they had participated in earlier Challenges. Thus, we aimed for a sense of continuity of the Challenge, to allow previous participants to re-use some of their previous investments, and for a sense of openness, to allow new participants to join the Challenge even if they

could not comply with the suggested targets (ARM7 or MPC) or programming language (C). We also followed the suggestion of the WCC'08 report to include PapaBench, already used in WCC'06 but not in WCC'08, again in WCC'11. We are thus happy to have had ten participating tools, up from five in 2006 and six in 2008. The ten 2011 participants include three 2006 participants (aiT, Bound-T, and SWEET), three 2008 participants (Bound-T again, OTAWA, and TuBound) and five first-time participants (Astrée, FORTAS, METAMOC, TimeWeaver, and WCA).

One price of the openness is reduced comparability of results. Ultimately, WCET analysis is about numbers, which should supply a natural metric to compare the tools. However, the presence of numerical results may also give a false sense of objectivity, and may tempt to compare apples with oranges. All participants provided numerical results, but these involved a range of target architectures, tool chains, and manual annotation effort. For future editions of the benchmark, it would be nice if more convergence could be reached here, at least for a "simple" processor/benchmark setting.

Furthermore, while we are convinced that all participants do their best to produce safe results (ie., to not underestimate the WCET), the absence of validated "true" WCETs also leaves the possibility of results that are (unintentionally) too good to be true. It is not clear how to circumvent this problem in practice. Then again, this is an issue that affects not only the WCC, but the whole WCET analysis discipline. Furthermore, the WCC might help the tool designers to uncover potential points for improvements in their tools (not only with respect to tightness, but also with respect to safety), which is just the point of the Challenge. Ideally, future editions of the Challenge would not only include safe estimates that strive for tightness and bound the true WCET from above (where lower is better), but would also include maximal established measurements that bound the true WCET from below (where higher is better). This still would not prove the safety of the WCET estimates, but could serve as a minimal consistency check.

One of the assets of WCC'11, the availability of an industrial code, also posed one of the organizational challenges. It turned out non-trivial to align the non-disclosure requirements and architectural constraints of the code with the capabilities of the participating tools. It would be nice if a future Challenge would have more participants for an industrial-size benchmark and the "complex processor" category.

The report on the last Challenge concluded [13]: "The WCC'08 organizers suggest that the Challenge should be defined as a continuous process, allowing the addition of benchmarks, participants, and analysis results at any time, punctuated by an annual deadline. At the annual deadline, a snapshot of the results is taken and becomes the result of the Challenge for that year." So far, this goal has turned out a bit too ambitious, but we hope with this Challenge to have made another step towards maturity of the Challenge and, more importantly, the involved tools. We certainly hope that there will be another WCC'1X Challenge, and hope that it will find a good balance between continuing established practice and adding new elements.

Acknowledgments From the Steering Committee, we wish to conclude by thanking all participants who actively contributed to the success of the Challenge from its very beginning, when they helped to define the setting, to the end, when they delivered their reports on time. We also thank the organizers of the previous Challenges, upon whose work we could build. Finally, we thank the chair of the WCET'11 Workshop, Chris Healy, for accommodating us concerning size and delivery date of this report.

References

[1] Advanced RISC Machines, ARM7DMI Data Sheet. Document Number ARM DDI 0029E, Issue E, Aug. 1995.

[2] BALLABRIGA, C., CASSÉ, H., ROCHANGE, C., AND SAINRAT, P. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, S. Min, R. Pettit, P. Puschner, and T. Ungerer, Eds., vol. 6399 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 35–46.

[3] BÜNTE, S., ZOLDA, M., AND KIRNER, R. Let's get less optimistic in measurement-based timing analysis. In *Proc. 6th International Symposium on Industrial Embedded Systems (SIES'11)* (June 2011). To appear.

[4] BÜNTE, S., ZOLDA, M., TAUTSCHNIG, M., AND KIRNER, R. Improving the confidence in measurement-based timing analysis. In *Proc. 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)* (Mar. 2011).

[5] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.

[6] DALSGAARD, A. E., OLESEN, M. C., TOFT, M., HANSEN, R. R., AND LARSEN, K. G. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)* (Dagstuhl, Germany, 2010), B. Lisper, Ed., vol. 15 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 113–123. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[7] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[8] GUSTAFSSON, J. The worst case execution time tool challenge 2006. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 233–240.

[9] GUSTAFSSON, J. WCET Challenge 2006. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-206/2007-1-SE, Mälardalen University, Jan. 2007.

[10] GUSTAFSSON, J., BETTS, A., ERMEDAHL, A., AND LISPER, B. The Mälardalen WCET benchmarks — past, present and future. In *Proc. 10^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2010)* (Brussels, Belgium, July 2010), B. Lisper, Ed., OCG, pp. 137–147.

[11] GUSTAFSSON, J., ERMEDAHL, A., LISPER, B., SANDBERG, C., AND KÄLLBERG, L. ALF – a language for WCET flow analysis. In *Proc. 9^{th} International Workshop on Worst-Case Execution Time Analysis (WCET'2009)* (Dublin, Ireland, June 2009), N. Holsti, Ed., OCG, pp. 1–11.

[12] GUSTAFSSON, J., ERMEDAHL, A., SANDBERG, C., AND LISPER, B. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06)* (Dec. 2006).

[13] HOLSTI, N., GUSTAFSSON, J., BERNAT, G., BALLABRIGA, C., BONENFANT, A., BOURGADE, R., CASSÉ, H., CORDES, D., KADLEC, A., KIRNER, R., KNOOP, J., LOKUCIEJEWSKI, P., MERRIAM, N., DE MICHIEL, M., PRANTL, A., RIEDER, B., ROCHANGE, C., SAINRAT, P., AND SCHORDAN, M. WCET Tool Challenge 2008: Report. In *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis* (Dagstuhl, Germany, 2008), R. Kirner, Ed., Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[14] HOLSTI, N., LåNGBACKA, T., AND SAARINEN, S. Using a Worst-Case Execution Time Tool for Real-Time Verification of the Debie Software. In *Data Systems in Aerospace (DASIA 2000)* (Sept. 2000), B. Schürmann, Ed., vol. 457 of *ESA Special Publication*.

[15] HOLZER, A., SCHALLHART, C., TAUTSCHNIG, M., AND VEITH, H. Fshell: Systematic test case generation for dynamic analysis and measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)* (Princeton, NJ, USA, July 2008), vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 209–213.

[16] HOLZER, A., SCHALLHART, C., TAUTSCHNIG, M., AND VEITH, H. Query-driven program testing. In *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)* (Savannah, GA, USA, Jan. 2009), N. D. Jones and M. Müller-Olm, Eds., vol. 5403 of *Lecture Notes in Computer Science*, Springer, pp. 151–166.

[17] HOLZER, A., SCHALLHART, C., TAUTSCHNIG, M., AND VEITH, H. How did you specify your test suite? In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)* (Sept. 2010).

[18] HOLZMANN, G. The power of 10: rules for developing safety-critical code. *Computer 39*, 6 (June 2006), 95–99.

[19] HUBER, B., PUFFITSCH, W., AND SCHOEBERL, M. Worst-case execution time analysis driven object cache design. *Concurrency and Computation: Practice and Experience* (2011).

[20] INFINEON. *TriCore Compiler Writer's Guide*. `http://www.infineon.com`, 2003.

[21] INFINEON. C167CR/SR Data Sheet, 2005. `http://infineon.com`.

[22] INFINEON. *TriBoard TC1796 Hardware Manual*. `http://www.infineon.com`, 2005.

[23] INFINEON. *TC1796 User's Manual V2.0*. `http://www.infineon.com`, 2007.

[24] KALIBERA, T., PARIZEK, P., MALOHLAVA, M., AND SCHOEBERL, M. Exhaustive testing of safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)* (New York, NY, USA, 2010), ACM, pp. 164–174.

[25] KNOOP, J., KOVACS, L., AND ZWIRCHMAYR, J. An Evaluation of WCET Analysis using Symbolic Loop Bounds. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)* (Porto, Portugal, July 5, 2011). To appear.

[26] KNOOP, J., KOVACS, L., AND ZWIRCHMAYR, J. Symbolic Loop Bound Computation for WCET Analysis. In *Proceedings of the 8th International Andrei Ershov Memorial Conference—Perspectives of System Informatics (PSI 2011)* (Akademgorodok/Novosibirsk, Russia, June 27–July 1, 2011). To appear.

[27] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Notices 30* (Nov. 1995), 88–98.

[28] LISPER, B., ERMEDAHL, A., SCHREINER, D., KNOOP, J., AND GLIWA, P. Practical experiences of applying source-level WCET flow analysis on industrial code. In *Proc. 4^{th} International Symposium on Leveraging Applications of Formal Methods (ISOLA'10), Part II* (Heraclion, Crete, Oct. 2010), T. Margaria and B. Steffen, Eds., vol. 6416 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 449–463.

[29] MICHIEL, M. D., BONENFANT, A., CASSÉ, H., AND SAINRAT, P. Static loop bound analysis of c programs based on flow analysis and abstract interpretation. In *RTCSA* (2008), pp. 161–166.

[30] NASA ENGINEERING AND SAFETY CENTER. Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation. Tech. rep., Technical Assessment Report, Dec. 2011.

[31] NEMER, F., CASSÉ, H., SAINRAT, P., BAHSOUN, J.-P., AND MICHIEL, M. D. Papabench: a free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis* (Dagstuhl, Germany, 2006), F. Mueller, Ed., Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[32] PITTER, C., AND SCHOEBERL, M. A real-time Java chip-multiprocessor. *ACM Trans. Embed. Comput. Syst. 10*, 1 (2010), 9:1–34.

[33] PRANTL, A., KNOOP, J., SCHORDAN, M., AND TRISKA, M. Constraint solving for high-level WCET analysis. In *Proceedings of the 18th Workshop on Logic-based Methods in Programming Environments (WLPE 2008)* (Udine, Italy, Dec. 12, 2008), pp. 77–89.

[34] PRANTL, A., SCHORDAN, M., AND KNOOP, J. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Post-Workshop Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (Prague, Czech Republic, July 1, 2008), vol. 237, Austrian Computer Society, pp. 141–148. Also: Schloß Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2008, ISBN 978-3-939897-10-1, 8 pages.

[35] RADIO TECHNICAL COMMISSION FOR AERONAUTICS. RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.

[36] RATSIAMBAHOTRA, T., CASSÉ, H., AND SAINRAT, P. A versatile generator of instruction set simulators and disassemblers. In *Proceedings of the 12th international conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems* (Piscataway, NJ, USA, 2009), SPECTS'09, IEEE Press, pp. 65–72.

[37] SCHOEBERL, M. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54/1–2* (2008), 265–286.

[38] SCHOEBERL, M., PUFFITSCH, W., PEDERSEN, R. U., AND HUBER, B. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience 40/6* (2010), 507–542.

[39] SOUYRIS, J., PAVEC, E. L., HIMBERT, G., JÉGU, V., BORIOS, G., AND HECKMANN, R. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th International Workshop on Worst-case Execution Time (WCET '05), Mallorca, Spain* (2005), pp. 21–24.

[40] TAN, L. The worst-case execution time tool challenge 2006. *Int. J. Softw. Tools Technol. Transf. 11* (Feb. 2009), 133–152.

[41] WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. P. Measurement-based timing analysis. In *ISoLA* (2008), pp. 430–444.

[42] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS) 7*, 3 (2008).

[43] ZOLDA, M., BÜNTE, S., AND KIRNER, R. Context-sensitivity in IPET for measurement-based timing analysis. In *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)* (Oct. 2010).