Lili Tan

The Worst-Case Execution Time Tool Challenge 2006

Abstract The first international Worst Case Execution Time (WCET) Tool Challenge in 2006 used benchmark programs to evaluate academic and commercial WCET tools. It aimed to study the state-of-the-art in WCET analysis. The WCET Tool Challenge comprised two parallel evaluation approaches: an internal evaluation by the respective tool developers and an external test by a neutral person of an independent institute. The latter was conducted by the author of this paper. Focusing on the external test, we describe the rules, benchmarks, participants and discuss the obtained results.

Keywords Timing Analysis · Worst Case Execution Time · WCET · Hard Real Time · Embedded Systems

1 Introduction

The Worst-Case Execution Time (WCET) denotes the longest execution time that a software program could take to run on a target processor under the worst case timing scenario. WCET information is required for schedulability analysis in safety-critical hard real-time systems, like flight control or automatic brake control systems, where timing constraints must be guaranteed.

Industrial demand on WCET analysis increases, since embedded control units and embedded devices are becoming more complex and pervasive. Therefore WCET analysis has received a lot of attention in recent years. This research results in a broad range of different approaches to WCET analysis and the development of various WCET analysis tools, which predict the WCET upper bounds.

To avoid isolated results within the research community and to make the area more transparent to industry, there has recently been a joint effort to compare the different approaches. An overview of the area of WCET analysis and a description of commercial tools for WCET analysis as well as research prototypes was published in a recent paper [43]. The focus of that work was to give a comprehensive overview, based on contributions from members of the community, and did not aim for an experimental evaluation.

The work presented here is based on a complimentary approach, an experimental evaluation of different WCET tools. The Challenge Working Group, organized by Jan Gustafsson of Mälardalen University, selected the benchmark programs, decided on the test rules and assigned the author of this paper to carry out the evaluation. While [36] concentrates on the design of the Challenge and summarizes test results obtained by tool developers, this paper focuses on the independent evaluation. It provides an in-depth analysis and interpretation of the results, extending a previous paper [40].

Outline. The remainder of the paper is organized as follows. We introduce background information on WCET and WCET analysis in Section 2. Section 3 lists the Challenge rules and Section 4 give a short introduction of the participating tools. The benchmark programs are described in Section 5. Section 6 explains how the tests were carried out, and Section 7 presents the detailed results of the tests and. Section 8 discusses the results. Section 9 concludes and gives an outlook on future work.

Lili Tan
ICB/Computer Science, University of Duisburg-Essen
Dependability of Computing Systems
Tel.: +49 -(0) 201-183 2340
Fax: +49 -(0) 201-183 2149
E-mail: lili.tan@icb.uni-due.de

2 Background

2.1 Worst-Case Execution Time

Hard real-time systems require an upper bound on the execution time of a program. Such a bound ensures that

even in a worst-case scenario the system will meet its deadlines. These bounds should be tight, i.e. as close to the worst-case as possible, to avoid unnecessary and costly safety margins. When determining such bounds, the input-dependence of execution time is one dimension of complexity. Depending on the inputs, e.g. start-up configuration variables for operational modes, or online sensor readings, the running time can vary significantly.

WCET analysis does not try to solve the halting problem. It is assumed that upper bounds on the number of times a loop is traversed, so-called loop bounds, and bounds on recursion depth can be determined either automatically or by manual annotation. Determining sufficient program control-flow information alone is a challenging problem. However, the hardware architecture also has a significant impact on execution time.

Processors employed in hard real-time systems range from micro-controllers such as the Infineon C16X processors and the Motorola HCS12 processor to more complex processors such as the PowerPC 565 (and its successors). Features of current processors like caches lead to a high variability of the execution time of instructions, e.g. a memory access that can be served from the cache is significantly cheaper than an access to main memory. In addition, superscalar processors execute several instructions of a sequential program in parallel in an instruction pipeline. The state of the pipeline, occupation of processor units, contents of buffers, the cache content, and peripherals all influence execution time and make execution time highly sensitive to execution history. This makes WCET analysis a challenging task.

2.2 Industrial Expectation

To enable widespread use of timing analysis in the industry, the timing analysis tools have to meet several requirements.

1. *Fundamental requirements*: To be applicable at all, a tool must be able to analyze the programs. Analyzability depends on the structure of the program as well as on the target processor and the compiler used to generate the binary code, all of which have to be supported by the tool.
   Also, the WCET bound must be correct, that is, the result must be a safe approximation of the actual worst-case execution time: The tool may over-estimate, but must not under-estimate.
2. *Quality requirements*: To be applicable in a practical setting, the computed bound should be tight: The over-estimation should be small.
3. *Usability requirements*: Usability refers to the extent to which a product [18], e.g., a software tool, can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. A one-click solution is always desirable. Although this is in general infeasible, a tool

should come as close as possible to this goal. The user should not be required to have specialist knowledge about WCET technology in general or the specific tool in order to use the tool and achieve satisfying results. Thus, the technology should be encapsulated inside the tool. If user effort is required to improve analysis results, the user should receive assistance by the tool in providing necessary annotations.

4. *Integration requirements:* Furthermore, the tool needs to be integrated into the software development process. To this end, input format and output format as well as intermediate results should be supported in the standard development process. For example, a WCET result which should be used in schedulability analysis has to be delivered in a suitable, possibly standardized format.

Under all these constraints, a tool should provide its result in a tolerable time.

2.3 WCET Tool Design

A WCET analysis tool takes as input a program and a description of the target architecture. Such a tool usually runs on a host computer and produces an upper bound on the execution time of the program on the target architecture, which often is different than the architecture of the host computer, i.e. the analysis software does not require access to the physical target architecture.

The standard tool architecture is to separate WCET analysis into three phases: a control-flow analysis phase, a micro-architectural phase, and an end phase combining the total effects resulting from both the target processor-dependent and independent analyses.

Control-Flow analysis phase. Most WCET tools analyze executables of the programs. In order to support subsequent analyses, the first step is to reconstruct the control flow graph from the executable. The control-flow graph represents a superset of the possible execution paths of the program that might be traversed during its execution. The complexity of that step depends on the structure of the code, and thus of the input program and the compiler. In particular, unstructured code, e.g., breaks, and multiple returns in a loop body and unconditional jumps (gotos) make the analysis more complicated. WCET tools are typically geared towards a particular compiler, so that control flow reconstruction can recognize compiler-specific patterns.

Some paths through the control flow graph do not correspond to actual program executions, e.g. a path that traverses two conditions that exclude each other is an admissible control flow path. These paths are called infeasible paths.

Information about infeasible paths, loop bounds and even about control flow can be determined by static program analysis techniques. Static program analyses obtain

information about all runs of a program without executing it. Abstract interpretation is a theory in which static program analyses that refer to deep semantics properties can be expressed.

Value analysis uses abstract interpretation to determine information about the values of registers and memory locations. This information can be used for control flow reconstruction, e.g. to resolve targets of jumps. Furthermore, a value analysis step is necessary for establishing memory access locations as well as loop bounds. Furthermore, certain infeasible paths can be discovered through value analysis.

Precise WCET analysis requires detection of loop bounds. This is complicated by program features such as nested loops, and bit operations on loop control variables. Due to the importance of loop bounds for WCET analysis and the complexity (in general, infeasibility) of accurate loop bound detection, this often requires users to provide such information as annotations.

If the processor does not feature a dedicated floating point unit, such computations have to be implemented by software. This complicates the analysis because loop bound detection for such code is particularly difficult.

Micro-architectural Phase. The description of the architecture is an executable model that keeps track of the micro-architectural features of the processor. Depending on the target processor, this may comprise the pipeline behavior, the state of the cache, the branch prediction unit, the translation lookaside buffer, behavior of a floating point unit, the memory hierarchy and the bus.

For a completely accurate WCET determination, all of the processor internals would have to be taken into account. In practical tools, an abstraction has to take place to keep analysis cost manageable. This abstraction can decrease the precision of the analysis.

Combined Calculation Phase. In many static WCET analyzer tools, the final phase is to combine the results of program control-flow analysis and the hardware micro-architectural timing effects by using Integer Linear Programming (ILP), which yields a WCET bound for the program.

Usability and Integration. Due to the complexity of the problem, WCET tools do not always offer a push-button solution. In some cases, the run time of the tools and the precision can significantly benefit from annotation. Furthermore, in case of analyses not being able to automatically determine facts such as loop bounds, the user is required to provide these as annotations. A tool's methods of demanding such information and of hinting the user to annotations improving the results form a large part of its usability.

The WCET tool must be designed appropriately to enable a tight integration into standard tool chains. This means that its inputs and outputs must fit into the standard development process.

2.4 Validation and Evaluation

All of the WCET tools are software products, and as such may be subjected to verification and validation. While the tool providers themselves may verify their tools, an external user is often not able to carry out a detailed validation. The reason is that the tools work as a black box, only outputting certain results. These cannot always be easily mapped to the work phases, and as such a complete validation is not possible from the outside.

However, we can evaluate the tools according to the expectations of the industry. In this context, degree of automation, support for certain language constructs, usability of tools, and precision of computed WCET bounds are important evaluation criteria. This is the goal of the WCET Tool Challenge, as detailed in the next sections.

3 Challenge Rules

3.1 The Challenge

A WCET Tool Challenge was discussed at the WCET Workshop 2006, a satellite workshop of the 18th Euromicro Conference on Real-Time Systems (ECRTS 06). For completeness, we briefly summarize the main goals of the Challenge (a more detailed mission statement can be found in [36,32]):

– collect a common set of benchmarks
– develop metrics for evaluation
– spur development of WCET analysis tools
– connect developers of tools.

The Challenge announcement with a call for participation was published in August 2006 in [32]. It was also distributed to known WCET research groups and companies in the world as well as to a WCET discussion group [30]. The Announcement introduced the requirements and rules for the competition, proposed benchmarks, and specified the deadline for registration and the publication of the test results.

WCET tool providers were requested to submit registration data to the organizer of the Challenge, Jan Gustafsson, by email before August 31st 2006. The registration data included information about tool names, the supported target processors and compilers.

Tool providers were given the opportunity to run their tool on the Challenge benchmark programs and report their results to the Challenge organizer [36].

Furthermore, an independent test person, who carried out tests on the same Challenge benchmarks, evaluated all of the tools on-site with some assistance by the tool developers. This is the external test, on which we focus in this paper.

3.2 Requirements

Requirements were rather liberal, in order to encourage participation. All WCET research groups or companies were allowed to participate in the Challenge [32]. All research prototypes or commercial products were allowed to be enrolled. Any tools using static analysis or measurement-based approaches as well as hybrid methods were welcome to participate.

No limitation on either the target processor or the compiler was imposed. Since a consensus on a single target processor was not to be expected, suggestions were made in the Announcement [32] with an effort to make the Challenge test results comparable. The suggested target processors range from simple processors (e.g., Renesas H8), to processors like the ARM7/9, C167, NEC V850E, and also includes very complex processors like the PowerPC 565.

Participants were allowed to submit variants of their tools for multiple target processors so that it was more likely to find common target processors supported by different participants, in order to make the test comparable.

Participants were free to use their compiler of choice and provide the resulting executable files to the test person. This was done to avoid licence problems and to admit tools that rely on information from a special compiler.

3.3 Benchmark Selection

Three suites of publicly available real-time benchmarks, the Mälardalen WCET benchmarks [19], PapaBench [22] benchmarks, and MiBench benchmarks, were originally proposed. A few days before the on-site tests, two of them were selected and announced to tool providers by the Challenge Working Group: The Mälardalen WCET benchmarks and PapaBench benchmarks. Some modifications were necessary to remove hardware-dependent code within PapaBench. The Working Group made the modified version available to all participants for the Challenge test. MiBench was not considered because it uses dynamic memory allocation, which is not suitable for safety-critical application.

3.4 The Three-Round Test Procedure

To assess the automation features and precision of tools, the test was run in a three-round fashion: without manual annotation, with minimal set of annotations, and finally with an optimal set of annotations to improve the WCET quality to the best level [32].

1. In the first round, no manual annotation was used to assess the capability of each tool to solve benchmarks automatically.

2. Some tools failed in the first round on certain benchmarks. In the second round, we began with a minimum amount of annotations, which made the tools run without claim for tightness. Such annotations include the loop bounds and iteration depth of recursions that a program required. We investigated the effects of adding more and more program constraints in annotations on precision and analyzability.
3. In the final-round, we fine-tuned the annotations to obtain optimal tightness, with support by the tool developers.

4 Participating Tools

Five WCET tool providers registered their tools for both the self-evaluation and the external test. We briefly describe the participating tools in the order of their registration.

4.1 aiT

aiT WCET Analyzer (aiT) is a software product of the company AbsInt Angewandte Informatik GmbH, Germany [1]. The technology used in aiT was initially developed at Saarland University [35]. Since 1998, AbsInt and Saarland University have been developing aiT jointly. The following AbsInt tools took part in the Challenge:

- aiT for Infineon C16X and the Altium Tasking compiler (version V8.5r1) [2]
- aiT for ARM7 (TMS470) and the Texas Instruments TI compiler (version v2.17) [29]
- aiT for PowerPC MPC565 and the WindRiver Diab compiler (version v5.3) [31]

We give a short description of the microarchitectures of these target processors:

- The Infineon C16x [17] is a family of 16-bit RISC micro-controllers with a classical four-stage pipeline (Fetch, Decode, Execute and Writeback). It has no floating-point unit. It features an on-chip flash memory to store the program. The C167 features a CAN-Bus controller which makes it suitable for the automotive domain.
- The TI TMS470 (ARM7TDMI core) [29] is an implementation of the 32-bit RISC ARM7DMI architecture. ARM7TDMI has a three-state pipeline and is the current low-end model of the ARM family. It is used as a component in system-on-chip solutions for mobile phones and portable communication and multimedia devices.
- The MPC565 [10] is a 32-bit PowerPC embedded microprocessor typically clocked between 40 and 66 MHz and deployed in automotive applications for engine and transmission control. It has an integrated floating point unit and one megabyte of FLASH memory on a single chip.

Table 1 Participating WCET Tools, Supported Target Processors and Compilers

Nr	Tool Name	Processor	Compiler	Affiliation
1	aiT	Infineon C16x	Tasking v8.5	AbsInt Angewandte Informatik GmbH, Germany
2	aiT	ARM7 TMS470	TI v2.17	AbsInt Angewandte Informatik GmbH, Germany
3	aiT	PPC MPC565	Diab v5.3.1.0	AbsInt Angewandte Informatik GmbH, Germany
4	Bound-T	Renesas H8/300	GCC, IAR	Tidorum Ltd., Finland
5	Bound-T	SPARC ERC32	GCC-based BCC	Tidorum Ltd., Finland
6	MTime	Motorola HCS12	COSMIC	TU-Vienna, Vienna, Austria
7	SWEET	ARM9	CtoNIC	Mälardalen University, Västerås, Sweden
8	Chronos	SimpleScalar *	GCC	National University of Singapore, Singapore

PPC = PowerPC.
* = SimpleScalar is a processor simulator [25]

Among the target processors supported by aiT , MPC565 is a very complex processor, according to the classification defined in the Challenge.

4.2 Bound-T

Bound-T is a software product of the company Tidorum Ltd. in Finland [5]. The following Tidorum tools participated:

- Bound-T for Renesas H8/300 and the GNU GCC C compiler [12] and IAR compiler [16], respectively
- Bound-T for SPARC/ERC32 and the Bare C Cross (BCC) compiler [11]

Here is a short description of the target processors:

The H8/300 is an 8-bit /16-bit micro-controller[14]. It is a simple processor which is not pipelined. It has no hardware floating point unit. Therefore, floating point calculations are implemented in software. Application domains of the H8 core include real-time control applications for automotive subsystems and toys.

Several compilers (e.g., IAR, GNU GCC) support the H8/300.

The SPARC/ERC32 is an implementation of SPARC V7 [26] processor. It is a 32-bit RISC processor. It has three devices; the integer unit (IU), the floating-point unit (FPU), and the memory controller (MEC). The SPARC/ERC32 has no cache. It is a radiation tolerant processor developed for space applications.

Derived from the GNUGCC compiler, the BCCcompiler has been adapted to the SPARC/ERC32 target by Gaisler Research.

4.3 SWEET

The SWEdish Execution time Tool (SWEET) is a research prototype developed at Mälardalen University [28]. SWEET supports the target processor ARM9 core [3] using the CtoNIC compiler [43].

The ARM9 core is an integer-only processor core. It has a 32-bit RISC CPU architecture featuring a five-stage pipeline. It has a Harvard memory architecture. There are separate buses for instructions and data. The clock speeds of the ARM9 cores are configurable. The ARM9 can be equipped with separate instruction and data caches. However, SWEET does not model cache behavior. It assumes a perfect memory system with single-cycle access to instructions and data [34]. The ARM9 core is used in mobile phones, routers, and other communication devices.

The CtoNIC compiler is a research prototype. In order to carry out the WCET analyses, SWEET depends on the intermediate code produced by CtoNIC.

4.4 MTime

MTime is a research prototype from the Real-Time Systems Group at Vienna University of Technology (TU-Vienna) [42] and [41]. MTime supports the processor Motorola HCS12 [15] with the COSMIC compiler [8]. We skip a detailed introduction about the target processor and compiler due to reasons stated in the Section 6. Still, for completeness, we take account of all registered tools that entered the Challenge.

4.5 Chronos

Chronos is a research prototype developed under GNU GPL license at the National University of Singapore. The underlying techniques of Chronos are described in [7].

Chronos is designed for SimpleScalar and the GCC compiler [12]. SimpleScalar is a research simulator suite for superscalar processors [25]. It implements a MIPS [21] instruction set and admits customization of several hardware parameters. The Chronos tool makes simplifying assumptions, e.g. a memory load operation completes in a single cycle.

Chronos entered the Challenge with three different target configurations:

- Simple in-order (SIO), in-order pipeline with perfect cache and perfect branch prediction
- Complex in-order (CIO), in-order pipeline with cache modeling and branch prediction

– Complex out-of-order (COO), out-of-order pipeline with cache modeling and branch prediction

4.6 Summary

We have introduced the five participating tools and the supporting processors and compilers. Table 1 summarizes the registration data.

Note that the target processors (and compilers) supported by the participating tools vary from each other. This has an impact on the comparability of the obtained results:

– impact of the target processor
  – The same executable has a different timing behavior on different target processor. Therefore the absolute numbers of the obtained WCET values are not comparable.
  – Beside obvious differences imposed by the instruction set, a different target processor may even lead to different control flow. As an example, consider functionality like certain floating point computations (e.g. square root, sine or cosine). If the processor does not provide a hardware floating point unit, such a computation is realized as a library function whose timing behavior is harder to analyze than an instruction realized in hardware.
– impact of the compiler
  – On the same target processor, different compilers or using the same compiler with other options typically leads to a different executable. Optimization options often affect control flow in connection with loops (such as loop unrolling and software pipelining) and conditionals (code motion and duplication). This may have an effect on loop bounds, infeasible paths and analyzability in some cases, although, at a coarse-grain level, control flow is preserved.
    Since the considered WCET tools are run on different executables (see Tables 4, 5, 6, 7), one has to be careful when comparing loop bounds and infeasible paths discovered by different tools.
  – Differing executables also make the absolute values of WCET obtained by tools running on different executables incomparable.

Although absolute WCET values are incomparable, degree of automation, support for certain language constructs and amount of overestimation (which is a relative value) are comparable.

5 The Benchmark Program Features

We introduce the selected benchmarks and discuss their features in language constructs, which are designed to evaluate WCET approaches and tools.

5.1 Mälardalen WCET Benchmarks

The Mälardalen WCET benchmarks are a collection of C programs from different sources:

– Programs developed at Mälardalen University (MDH)
– SPEC95 [27]
– Seoul National University Real-Time Research Group (SNU-RT) [38]
– C-Lab, Paderborn [6]

The benchmarks are maintained at Mälardalen University and were originally used to validate SWEET.

The MDH benchmark programs are synthetic, intentionally hard, test cases for flow analysis, e.g.,

– `cnt`: Count non-negative numbers, contains nested loops
– `cover`: Loop containing many switches
– `duff`: Duff's device jargon file copy, contains jump into loop body
– `edn`: JPEG compression, contains bit shift operation
– `janne_complex`: Nested loops, contain many infeasible paths
– `malmult`: Matrix multiplication, contains nested for-loops
– `ndes`: Complex embedded code, contains counted loop and bit operations
– `ns`: Search in a multi-dimensional array, deeply-nested loop
– `recursion`: Recursive Fibonacci function

Benchmark program `compress` was taken from SPEC95. It is a hand-coded program for data compression, containing bit operations and loops for shifting.

Benchmarks originally from SNU-RT are hand-coded algorithms from real-time applications:

– `adpcm`: Adaptive Differential Pulse Code Modulation, contains variable multiplication in loop range expression
– `crc`: Cyclic redundancy check, contains loop bounds depending on function arguments
– `insertsort`: Insertion sort, contains a nested loop with a complex dependence between iteration variables

The C-Lab Benchmarks are automatically generated programs from the embedded real-time domain:

– `nsichneu`: Simulates an extended Petri net, state transitions realized using if-statements
– `statemate`: Car window lift control generated from a Statechart [37], state transitions realized by switch statements

Each of the 15 selected Mälardalen WCET benchmarks consists of one single source file. We have summarized some properties of the benchmark programs in Table 2[1].

---

[1] The table extends the table from Mälardalen WCET benchmarks [19] with information about language constructs for each program.

Table 2 Benchmarks: Description, Source Code Size, and Features

Nr.	Benchmarks	Description	Source	Type	Size (LOC)	Size (KB)	L	N	A	B	R	U
1	adpcm	ADPCM Signal-processing	SNU-RT	H	879	27	L		A	B		U
2	cnt	Count non-negative numbers	MDH	H	267	4	L	N	A			
3	compress	Data compression	SPEC95	H	508	13	L	N	A	B		
4	cover	Test many paths	MDH	H	240	7	L					
5	crc	CRC computation	SNU-RT	H	128	6	L		A	B		
6	duff	Duff's Device	MDH	H	86	3	L		A			U
7	edn	DCT JPEG compression	MDH	H	285	11	L	N	A	B		
8	insertsorts	Insertion sort	SNU-RT	H	92	5	L	N	A			
9	janne_complex	Nested loops	MDH	H	64	2	L	N				
10	matmult	Matrix multiplication	MDH	H	163	4	L	N	A			
11	ndes	Complex embedded code	MDH	H	231	8	L		A	B		
12	ns	Multi-dimensional array	MDH	H	535	10	L	N	A			U
13	nsichneu	Petri-Net Simulation	C-Lab	S	4253	105	L		A			
14	recursion	Fibonacci recursion	MDH	H	41	1					R	U
15	statemate	Car window lift control	C-Lab	S	1276	42	L		A	B		
16-17	PapaBench Package	Fly-by-wire and autopilot control	IRIT	H	**2000**	**269**	**L**		**A**	**B**	**R**	**U**

LOC = Lines of source code.
ADPCM = Adaptive Differential Pulse Code Modulation.   CRC = Cyclic redundancy check.
DCT = Discrete Cosine Transformation.
SNU-RT = Seoul National University, Real-Time Research Group   MDH = Mälardalen University.
S = Synthesized automatically by a code generator.   H = Hand-written code.
L = Loops. N = Nested loops. A = Array/Matrices. B = Bit Operation. R = Recursion. U = Unstructured code.

5.2 PapaBench Benchmark Package

Derived from the Paparazzi project [23], the PapaBench benchmark is developed for a real real-time embedded application. It is designed to be embedded on different Unmanned Aerial Vehicles (UAV) for autonomous aircraft control and fly-by-wire. The software is available from the IRIT Institute [22].

Tasks in PapaBench are embedded in a real system with hard timing constrains. WCET bounds are needed to schedule tasks for two applications:

- autopilot, an autonomous aircraft control based on a flight plan
- fly-by-wire, servo control and communication with autopilot

The PapaBench programs use code from different source files, unlike Mälardalen benchmarks where each program is contained in a single file. The whole software package consists of 22 folders and 120 files. Therefore we summarize all the language constructs for the whole PapaBench package in Table 2 without separating them into single files.

5.3 Summary

We have presented the selected benchmarks which range from academic benchmarks to tasks from hard real-time applications, which includes both hand-coded algorithms as well as control-intensive programs created by code generators of model-based development environments, e.g. Statemate. The selection includes problems from the automotive and the aeronautic domain.

6 The Experiments

We aimed at giving each tool a fair share of time spent on its evaluation. A schedule had been made to visit the tool developers and to test their tools (see Table 3). Roughly one week was spent on each tool (including traveling). The tests were performed starting with aiT and continuing with MTime, Bound-T, SWEET, and Chronos (in First In First Service order).

We skip MTime because MTime was not able to analyze function calls at that time. All selected benchmarks contain function calls. Therefore MTime could not deliver results in the Challenge.

The main computer used to install the participating tools and to run the external tests is a laptop with an AMD Mobile Sempron TM Processor 2800+ with 1.60 GHz and 448 MB RAM.

Three additional computers were used at the University of Duisburg-Essen for testing Bound-T and Chronos. These computers have AMD Athlon TM 2500+ CPUs at 1.83 GHz, and 992 MB of RAM. Two ran Windows XP and one Linux. For Bound-T, the PCs ran in parallel, because it took over two days for Bound-T to analyze the benchmark `autopilot` without annotations. For technical reasons, the PCs were used to access Chronos via a remote server in Singapore.

Table 3 Tool-Testing Schedule

Nr.	Duration	Test Activities	Place
0	2006-09-29	Start right after the organizational decision made	Essen, Germany
1	2006-10-02 to 06	Test aiT, aiT trip (2006-10-03 to 06)	Saarbrücken, Germany
2	2006-10-09 to 13	MTime trip (2006 -10-10 to 13)	Vienna, Austria
3	2006-10-16 to 20	Test aiT, Test Bound-T	Essen, Germany
4	2006-10-23 to 27	Test Bound-T, Bound-T and SWEET trip (2006-10-24 to 28)	Västerås, Sweden
5	2006-10-30 to 11-03	Test Chronos, Test Bound-T, Test SWEET	Essen, Germany
6	2006-11-06 to 13	Report drafting, Test SWEET	Essen, Germany
7	2006-11-14 to 20	ISoLA conference trip, Test report on 2006-11-17	Paphos, Cyprus

Table 4 aiT's Input Programs: Format and Size (KB)

Nr.	Benchmarks	aiT C16X Tasking	aiT ARM7 TI	aiT MPC565 Diab
	Code Format	*.abs	*.out	*.elf
1	adpcm	29	30	19
2	cnt	5	8	7
3	compress	16	17	11
4	cover	15	8	9
5	crc	6	8	7
6	duff	4	7	6
7	edn	15	16	10
8	insertsorts	3	5	6
9	janne_complex	4	5	6
10	matmult	5	8	7
11	ndes	14	14	13
12	ns	6	11	11
13	nsichneu	102	73	41
14	recursion	4	6	7
15	statemate	32	33	26
16	fly-by-wire	34	36	31
17	autopilot	121	136	104

We begin the experiments section by describing how to work with the tools, e.g. which inputs are required, the input and output formats, and the user interfaces.

6.1 aiT

Input. There are three types of input for aiT:

- A program executable
- A specification file for machine settings and optional program annotations
- A program entry point (like a `main` function)

The employed executables were compiled by an aiT developer and were made available for the external test. We list the formats and sizes of these executables in Table 4.

As stated in Section 4, we observe that for each target platform there is an executable format and that, for a particular program, the size of the executable varies for the different targets, e.g. the benchmark `nsichneu` compiled by the compiler Tasking takes 102 KB in size, by the compiler Diab 41 KB, although the C source file is the same.

An aiT specification file includes information about the settings of the target processor, e.g., clock rate and properties of memory areas, and program annotations if required, e.g. loop bounds.

User Interaction. The aiT tool provides a graphical user interface (GUI) to manage files in a project (the input program, the specification file and output files), gives assistance to develop annotations, and presents analysis results.

User effort is required if loop bounds and recursion are not detected by the tool automatically. Providing annotations to eliminate infeasible paths can improve precision. aiT provides hints for necessary annotations and displays the position where the annotation is needed, in the position of the executable file as well as in the corresponding source code line (if source code is available).

Annotations may contain any of the following information about a program:

- targets of computed calls and branches,
- loop bounds if not automatically detected by the tool,
- recursion depth,
- code snippets that should not be analyzed,
- infeasible code.

Output. aiT produces report files and a graphical visualization. The report files (in txt and XML format) contain

- timing information: the calculated WCET value for the program and contributions of different parts of the program to this total value. Furthermore, calculated execution time bounds can be mapped back to the program code by inspecting the visualization.
- program analysis information, including value analysis.

The graphical visualization shows the control flow graph of the program enriched with timing information and inspect states of the processor model at desired program locations.

6.2 Bound-T

Input. There are three types of input for Bound-T:

- A program executable

Table 5 Bound-T Input Programs: Format and Size (KB)

Nr.	Benchmarks	Bound-T H8/300 GCC	Bound-T H8/300 IAR	Bound-T SPARC BCC
	Code Format	*.coff	*.iar	*.elf
1	adpcm	29	21	107
2	cnt	17	5	92
3	compress	24	12	98
4	cover	19	13	93
5	crc	16	4	92
6	duff	15	4	90
7	edn	24	12	97
8	insertsorts	15	2	90
9	janne_complex	15	3	90
10	matmult	17	5	92
11	ndes	23	12	96
12	ns	17	5	95
13	nsichneu	111	84	139
14	recursion	15	3	90
15	statemate	32	28	119
16	fly-by-wire	56	36	108
17	autopilot	320	157	187

Table 6 SWEET Input Programs: Format and Size (KB)

Nr.	Benchmarks	SWEET ARM9 CtoNIC	
	Code Format	*.nic	*.tcd
1	adpcm	928	623
2	cnt	70	59
3	compress	302	294
4	cover	723	639
5	crc	132	172
6	duff	110	82
7	edn	547	569
8	insertsorts	35	36
9	janne_complex	29	20
10	matmult	74	56
11	ndes	752	346
12	ns	527	37
13	nsichneu	3076	3148
14	recursion	34	23
15	statemate	853	908
16	fly-by-wire	N/A	N/A
17	autopilot	N/A	N/A

- A specification file for program annotations if required
- A program entry point to start the WCET analysis

There is no need to specify the target machine setting. Bound-T assumes a fixed machine setting. The executables were compiled with the respective commercial and non-commercial compilers by the Bound-T developer, who made them available for the external test. We list the formats and sizes of these executables in Table 5. We notice from the table that the code size of the benchmarks from BCC compiler is on average greater than those produced by the other two compilers, except for benchmark `autopilot`.

User Interaction. The main work of user interaction for Bound-T is to develop an annotation file. Bound-T is a command-line tool. It does not give explicit hints concerning necessary annotations. If the tool gets stuck at a particular program location, this may be an indication that an annotation, such as a loop bound, is necessary. To this end, runtime messages indicate the program location currently being processed.

There is no GUI or generic interface to trace back the appropriate positions in executable and in source files, where annotations are needed. Users have to do it by themselves.

Visualization of control flow graph is realized via the open source software Graphviz [13].

Output. Bound-T produces command-line and graphical output giving the calculated WCET bounds and some intermediate results of the benchmarks.

6.3 SWEET

Input. In addition to an executable, SWEET needs as input the intermediate code from a specific compiler, the CtoNIC compiler. The compiler was able to be compile the 15 Mälardalen WCET-benchmarks but failed in the two PapaBench benchmarks. Mälardalen University made the compiled 15 Mälardalen WCET-benchmarks available for the external test. The inputs for SWEET are summarized in Table 6 (programs that could not be compiled by CtoNIC are marked with N/A).

User interaction. SWEET comes with multiple analysis engines for control-flow analysis, called *modes* by the SWEET developers. Users need to pick an appropriate mode. Furthermore, one can specify interval ranges in which the value of a variables lies.

There are four modes available for SWEET:

- Simple path basic mode (SPBM): produce loop bounds without using interval information for variables
- Simple path advanced mode (SPAM): produce loop bounds and infeasible paths without using interval information
- Multi path basic mode (MPBM): produce loop bounds using interval information
- Multi path advanced mode (MPAM): produce loop bounds and infeasible paths using interval information

The MPBM and MPAM modes are not applicable to all programs. They only worked for `crc`, `edn`, `insertsort`, `janne_complex`, `ns`, and `nsichneu` of the Mälardalen WCET-benchmarks in the Challenge.

SWEET is a command-line tool that displays runtime messages giving calculated results in the Windows

Table 7 Chronos Input Programs for Core WCET Analysis: Format and Size (KB)

Nr.	Benchmarks	Chronos SimpleScalar GCC
	Code Format	
1	adpcm	82
2	cnt	68
3	compress	74
4	cover	N/A
5	crc	68
6	duff	N/A
7	edn	73
8	insertsorts	67
9	janne_complex	67
10	matmult	69
11	ndes	73
12	ns	72
13	nsichneu	114
14	recursion	N/A
15	statemate	81
16	fly-by-wire	80
17	autopilot	181

N/A = Chronos did not handle the benchmark.

console. There is neither hint for annotations nor for execution mode selection.

SWEET's visualization of control flow graphs is realized via the open source software Graphviz [13], like Bound-T. It displays hierarchical structures of nested loops.

Output. SWEET outputs runtime messages in the Windows console, which contain WCET bounds information that the tool calculates.

6.4 Chronos

Input. The Chronos tool chain starts with C source code. The SimpleScalar infrastructure comes with a GCC compiler. It produces an executable, which Chronos uses as input to the WCET analysis. We have compiled the benchmarks using the tool chain. The executables obtained for core WCET analysis of Chronos are summarized in Table 7.

Since Chronos supports three target configurations, i.e., SIO, CIO, COO. Each of the configuration settings was set in a file as input. This was necessary for both the estimated WCETs and the simulated execution times.

User Interaction. Loop bounds, which were not automatically detected by Chronos, were required to be provided in an annotation file.

There is a Chronos GUI available. However, we used the command-line interface instead during the Challenge. Because the response time of the remote access to the GUI was terrible. Since we had to access CPLEX [9],

a commercial ILP solver in Chronos work-flow for the *Combined Calculation Phase* described in Section 2.3.

Output. Outputs of Chronos include WCET results and statistics on analysis times.

6.5 Summary

We observe that requirements on the compiler, e.g. the need for a specific form of intermediate code, may complicate the adaption and integration into a development process.

It would be interesting to inspect the results of the intermediate phases. However, some tools are black boxes, i.e. they do not allow looking into results of the intermediate analysis phases. This also precludes a one-to-one comparison between tools at the level of intermediate phases such as flow analysis.

Due to the black-box character of some tools, we evaluate tools based on ability to analyze programs, level of automation and precision.

7 Test Results

We present test results from the three-round test procedure in this section. We discuss:

– The ability of WCET tools to analyze benchmarks automatically
– Human effort in improving WCET analyzability and tightness of results. We use tool facilities to provide tools with user annotations, in order to process WCET analysis and to improve prediction quality.
– The final results with full annotations:
  – total numbers of benchmarks analyzed
  – the prediction quality by means of WCET tools outputs vs. measurement

We present our observation and an interpretation in the summary.

7.1 The First-Round: Test Results without Annotations

Benchmarks Solved Automatically. In the first test round, we provided each tool with the benchmark programs in its required formats, without user interaction for annotation of the program features.

The results of the first round are given in Table 8 and in Table 9. We give the WCET in CPU cycles predicted by the different tools (in column) for the corresponding target processors (in column) and respective compilers (in column) for each benchmark program (in row). If a tool could not analyze a benchmark program, this is indicated by a blank in the table. The absolute WCET values in the table are incomparable due to different target processors. Nevertheless, we provide these

Table 8 WCET Values (in cycles) Predicted by Tools Without User Annotations

Nr.	Benchmarks	aiT C16X Tasking	aiT ARM7 TI	aiT MPC565 Diab	Bound-T H8/300 GCC	Bound-T H8/300 IAR	Bound-T SPARC BCC
1	adpcm						
2	cnt	32812	26572	7576	45806	78982	
3	compress						
4	cover	19459	6780	5451		10250	180
5	crc			107278	164118	268657	
6	duff						
7	edn		307889	104907			
8	insertsorts						
9	janne_complex						
10	matmult	1562815	523599	237736	1506520	3282132	
11	ndes	816337	194448	1944845		712454	4214
12	ns	238414	38043	34361	20976	256960	7097
13	nsichneu		41678	21362			
14	recursion						
15	statemate						
16	fly-by-wire test_ppm_task		9875	1242			
	fly-by-wire send_data_to_autopilot_task		3197	331			
	fly-by-wire check_mega128_values_task		4092	437			
	fly-by-wire servo_transmit	2390	1909	1249			
	fly-by-wire check_failsafe_task		4058	432			
17	autopilot radio_control_task		15972	2247			
	autopilot stabilisation_task		4239	340			
	autopilot link_fbw_send	170	144	81			
	autopilot receive_gps_data_tasks						
	autopilot navigation_task						
	autopilot altitude_control_task		915	95			
	autopilot climb_control_task		4129	247			
	autopilot reporting_task	8286	11172	4464			
1-17	Analyzed automatically	8	18	19	4	6	3

values, because they indicate how much precision was gained by annotation. We concentrate on the number of benchmarks and tasks each tool solved (the last line in the table), which indicates the degree of automation of a tool.

The benchmarks have different levels:

– All tools succeeded in automatically analyzing programs cnt and matmult. Program cnt counts non-negative entries in a matrix. It contains nested loops where loop bounds are easy to determine and consists only of well-structured code. The program matmult is matrix multiplication for 20x20 matrices and performs multiple calls to the same function. It contains nested function calls and triply-nested loops.
– Many tools could automatically handle cover, crc, edn, ndes, ns and nsichneu. It seems impossible to attribute this to any particular coding style or language construct.
– Among the harder cases are the following programs because of difficult control flow, namely, adpcm, compress, insertsort, janne_complex, and Duff's device duff. Program duff is a test case for unstructured code, adpcm is a milder form of unstructured code. Programs janne_complex and insertsort are difficult because they contain a nested loop, where

the inner condition depends on the outer loop variable.
– Because only aiT (with annotated recursion depth) and SWEET support recursion, these tools were the only ones which could analyze program recursion.
– PapaBench is a special case because only aiT could analyze automatically large parts of it (all but two). In particular, loop bounds of for-loops were discovered automatically. aiT required annotations of loop bounds for while-loops and loops from the floating-point library.

Above, we have taken a benchmark-oriented view (lines in Table 8 and Table 9), i.e. we have grouped together benchmarks according to how well the tools perform on them. We proceed by focusing on the tools (columns in Table 8 and Table 9) and provide specific detail for each tool:

– The aiT tools excelled on the application-oriented real-time control programs of PapaBench. This seems to be due to aiT's use of abstract interpretation which scales well on large programs while some others tools timed out.
Since the C16X target machine does not support floating point operations in hardware, these functions have to be implemented in software. These algorithms com-

Table 9 WCET Values (in Cycles) Predicted by Tools Without User Annotations

Nr.	Benchmarks	SWEET ARM9 CtoNIC (SPBM/SPAM)	Chronos SimpleScalar GCC (SIO)	Chronos SimpleScalar GCC (CIO)	Chronos SimpleScalar GCC (COO)
1	adpcm	2165650 / 2162122			
2	cnt	36719 / 35319	4896	6438	5401
3	compress	206480 / 49896			
4	cover	73128 / 63563			
5	crc	834159 / 830278			
6	duff	5525 / 4720			
7	edn	1425085 / 1425085	89401	113612	89030
8	insertsorts	31163 / 18167			
9	janne_complex	12039 / 2523	189	800	789
10	matmult	2532706 / 2532706	186903	191615	119526
11	ndes	795425 / 795425			
12	ns	130733 / 130631			
13	nsichneu	119707 / *			
14	recursion	29079 / 20033			
15	statemate	15964 / 8451			
16	fly-by-wire test_ppm_task				
	fly-by-wire send_data_to_autopilot_task				
	fly-by-wire check_mega128_values_task				
	fly-by-wire servo_transmit				
	fly-by-wire check_failsafe_task				
17	autopilot radio_control_task				
	autopilot stabilisation_task				
	autopilot link_fbw_send				
	autopilot receive_gps_data_task				
	autopilot navigation_task				
	autopilot altitude_control_task				
	autopilot climb_control_task				
	autopilot reporting_task				
1-17	Analyzed automatically	15	4	4	4

* = Memory exhausted in the test laptop.
SPBM = Single path basic mode. SPAM = Single path advanced mode.
SIO = Simple in order. CIO = Complex in-order. COO = Complex out-of-order.

pute approximations up to a particular precision and thus have complex termination criteria which require annotation because static analysis is unable to accomplish this automatically.

– SWEET was the only tool that solved all Mälardalen benchmarks automatically. However, it was unable to analyze the PapaBench examples. SWEET does not have a single generic engine, it rather has specialized modes for particular program constructs. In the table, we therefore list results for the two most appropriate modes.

– Chronos and SWEET were the only tools to automatically analyze `janne_complex`.

aiT and Bound-T entered the Challenge with more than one target processor. We observe that the target processor made a difference in terms of whether or not a program could be analyzed automatically, e.g. aiT for MPC565 and aiT for ARM7 were able to solve benchmarks more than aiT for C16x. This is because both the MPC565 and ARM7 are register-oriented processors whereas the C16x is a stack-based processor. In a stack-based architecture, local variables of functions are on the stack rather than in registers. Therefore, value analysis for a stack-based architecture needs to resolve dynamic memory addresses of stack variables, which makes value analysis harder than for a register-oriented architecture.

Bound-T for SPARC/ERC32 solved less benchmarks than Bound-T for the other two targets.

In the first round, some tools solved certain benchmarks automatically. In the next round, we tried to analyze the remaining benchmarks by providing the tools with manual annotations.

7.2 The Second-Round: Annotations and Effects

We make the following observations:

– In cases where benchmarks were not solved automatically, annotations were required for analyzability,
– In cases where benchmarks were not solved automatically, annotations improved test results,
– In some cases where annotations were allowed, certain tools could still not analyze all benchmarks.

The subsection is structured according to these different cases.

Annotations Improving Analyzability. What follows is a detailed description of the annotations that were required for successful analysis:

Some of the PapaBench programs ,e.g., `autopilot` require normalization of angles modulo of 360 degrees which involves floating point arithmetic and comparison of floating point values in a loop condition. These programs could be analyzed after providing manual loop bounds (see Table 10).

Loops and nested loops with dependencies between inner and outer iteration variables could be handled with manual annotation e.g., for benchmarks `compress`, `edn`, `insertsort`, and `janne_complex`. To recall this, SWEET detected these automatically for the Mälardalen benchmarks.

aiT was able to analyze the recursive program with help of annotation for the numbers of the recursive iterations.

Annotations Improving Tightness. aiT was able to provide some hints about loop bounds in its GUI. Using these messages for annotations improved the tightness of results. Additionally, specifying infeasible paths and memory access areas, further improved tightness.

SWEET uses different program modes to eliminate infeasible paths and reduce overestimation. By specifying variable ranges in case of benchmarks `crc`, `edn`, `insertsort`, `janne_complex`, `ns`, and `nsichneu`, the precision of SWEET was improved.

SWEET, Bound-T and Chronos did not give hints in terms of annotations for better tightness.

Even Annotations Do Not Help. There are general limitations in some tools. Because of the limitations, some tools could not analyze some benchmarks even with annotation:

Bound-T was still not able to handle a loop with multiple entry points, (such as in `duff`) and recursion (as in `recursion`). It was not able to analyze the nested loops in `janne_complex` despite allowance for annotation. Bound-T bailed out on benchmark `statemate` with an exception.

The compiler framework on which SWEET relies, the CtoNIC, was not able to compile two of the PapaBench-Benchmarks. As a result, no further analysis was possible for SWEET.

Chronos was not able to analyze the benchmarks [2] `cover`, `duff`, `recursion` and `autopilot`.

As a result, by providing annotation, the numbers of the benchmarks successfully analyzed increased and tightness improved generally. We report these quantitative results in the next section.

---

[2] A release of Chronos submitted after the Challenge succeeded on the `cover` and the `duff` benchmark.

7.3 Final: Test Results with Annotations

We count the total effects on the final results:

1. under the aspect of quantity, the number of analyzable test programs increased;
2. under the aspect of quality, the execution time bounds became tighter.

This subsection is structured as follows:

– Total benchmarks analyzed in the end results
– Precision by means of calculated WCET vs. measurements

Total Benchmarks Analyzed. Table 10 summarizes the number of analyzed cases with the help of annotation and gives the causes of failed analysis. In a nutshell, the total numbers of programs analyzed rose to 100%, which was achieved by aiT for all Mälardalen WCET-Benchmarks and PapaBench-Benchmarks. Bound-T succeeded with the PapaBench-Benchmarks, while the research tool SWEET was good at handling Mälardalen WCET-Benchmarks.

Now we explain problems encountered.

– Compilation problem: Some benchmarks could not be compiled. Therefore, no input was available for the WCET analysis. This applied to the CToNic compiler for SWEET in analyzing the PapaBench benchmark programs. The other WCET tools, which analyze code compiled by common GCC or commercial compilers, are not necessary to be restricted by the same problem.
– Analysis problem: This applied to the benchmark programs that could be compiled but could not be analyzed. Usually it is because of the limitation in tools and approaches. Some tools were not able to handle some language constructs. The challenging language construct include unstructured code. Recursion (as in `recursion`) was not supported by Bound-T nor by Chronos. The Omega Calculator [39], which Bound-T uses for loop bound calculation , was not able to analysis nested loops in the benchmark `janne_complex` and `statemate`.

For completeness, we report on minor problems such as resource limitations of the test machine and other technical problems:

– On MS Windows XP, there were compatibility issues with the Omega Calculator. This affects BOUND-T.
– Due to limited memory of the machine, aiT for ARM7 ran out of RAM on `matmult` in the third round. The same happened for SWEET on `nsichneu`. No such problems had been encountered on a larger computer reported by [36].

After annotation, aiT analyzed all benchmarks in all categories and SWEET analyzed all Mälardalen benchmarks. Bound-T and Chronos do not solve the MDH,

Table 10 Total Benchmarks Analyzed and Problems Encountered

Nr.	Benchmarks	aiT	Bound-T	SWEET	Chronos
1	adpcm				
2	cnt				
3	compress				
4	cover				f
5	crc				
6	duff		f		f
7	edn		e		
8	insertsorts				
9	janne_complex		f		
10	matmult	a			
11	ndes				
12	ns				
13	nsichneu		e	a	
14	recursion		f		f
15	statemate		f		
16	fly-by-wire			f	
17	autopilot			f	f
	Analyzed				
1-17	total (#)	17	13	15	13
1-17	ratio (%)	100%	76.5%	88.2%	76.5%

f =   Fatal error. The tool does not handle these problems with the release at that time.
e =   Error. Microsoft Windows informed that the Omega Calculator had a conflict with Microsoft Windows System.
a =   Memory exhausted: aiT for ARM7 on the test laptop, and SWEET in single-path advanced mode.
=   Numbers of the analyzed benchmarks.
% =   Percent.

SNU-RT, and C-Lab benchmarks completely. This is due to unsupported language constructs such as recursion, and in other cases because switch tables could not be reconstructed. This is because Bound-T and Chronos were unable to resolve indirect jumps in switch tables. The value analysis used in aiT was precise enough to resolve these addresses. Chronos did not implement a value analysis.

WCET Prediction vs. Measurement. In order to examine the correctness and precession of the computed WCET upper bounds by static analysis tools, we compared them to the tentative worst-case execution times observed by measurement. We would like to point out that the actual WCET of a program is often unknown. In general, it lies above the measured execution time and below the computed WCET estimate. Therefore, by comparison of the prediction and measurement values, we get an upper bound on the overestimation produced by a tool, since the difference between computed WCET estimate and measured execution time is greater that or equal to the difference between actual WCET and measurement.

Measurement were conducted for aiT and Chronos. For aiT, the execution times were measured on real hardware using a logic analyzer. Some programs did not fit into the trace memory of the logic analyzer. This is indicated by *buffer* in the table. For Chronos, the Sim-pleScalar simulator has been used. Here *N/A* indicates that Chronos could not analyze the program. PapaBench could not be executed due to hardware dependency. For the other participating tools, no reference values were available.

Table 11 gives the WCET values calculated by aiT and Chronos with annotations. Measured execution times are given in Table 12. The tightness with respect to these reference values is given in Table 13.

Precision in tightness of the analyzed results from aiT for ARM7 was 1.47%, for C16x 8.82%, for MPC565 12.6%. aiT obtained tighter results for the simpler target processor than for the more complex one.

The target for Chronos is the SimpleScalar processor simulator. The tightness for the simple in-order pipeline was 33.69%, for complex in-order pipeline it was 80,52%, and for complex out-of-order pipeline 131,54%. Chronos obtained tighter results for the simpler hardware configuration than the more complicated one: the difference may vary largely by up to a factor of four.

Here, the complexity of the target architecture influenced the precision of a tool.

Also, the overestimation for both tools was program-dependent, e.g., for benchmark `compress`, we observed a higher overestimation than for the other benchmark programs. This program contains bit-operations, many conditions, and loops for shifting.

Table 11 WCET Values (in Cycles) Predicted by Tools With Optimal Annotations

Nr.	Benchmarks	aiT C16x	aiT ARM7	aiT MPC565	Chronos SimpleScalar (SIO)	Chronos SimpleScalar (CIO)	Chronos SimpleScalar (COO)
1	adpcm	558342	1375886	430274	265588	347742	317354
2	cnt	20250	17053	7376	4896	6438	5401
3	compress	37570	20280	9461	5873	29215	28487
4	cover	10452	6780	5006	f	f	f
5	crc	275910	213337	98830	47786	61849	53275
6	duff	7196	4612	1355	f	f	f
7	edn	927068	307889	88381	89401	113612	89030
8	insertsorts	4870	3992	1838	901	1549	1245
9	janne_complex	1330	829	383	189	800	789
10	matmult	956710	a	237736	186903	191615	119526
11	ndes	453348	194448	130025	66655	107589	85918
12	ns	75712	38043	18215	8199	9991	8676
13	nsichneu	29840	18827	8327	13609	97908	97525
14	recursion	10076	7451	5527	f	f	f
15	statemate	2620	3812	1294	2007	16185	16103

f = Fatal error. The tool does not handle this problems with the release at that time.
a = Memory exhausted in the test laptop.

Table 12 Measured and Simulated Execution Times

Nr.	Benchmarks	aiT C16x	aiT ARM7	aiT MPC565	Chronos SimpleScalar (SIO)	Chronos SimpleScalar (CIO)	Chronos SimpleScalar (COO)
1	adpcm	buffer	buffer	buffer	160891	183526	126258
2	cnt	19622	16853	7235	4792	5586	3515
3	compress	27308	19970	6824	5859	7504	4744
4	cover	10080	6778	4299	N/A	N/A	N/A
5	crc	buffer	buffer	buffer	22688	26861	18098
6	duff	6919	4610	1028	N/A	N/A	N/A
7	edn	838686	299734	buffer	87444	108973	62995
8	insertsorts	4720	3990	1770	897	1364	949
9	janne_complex	1294	827	359	185	454	356
10	matmult	936602	438435	buffer	186899	185937	90834
11	ndes	401294	190530	buffer	65600	86639	53625
12	ns	73738	36097	buffer	6577	7568	4784
13	nsichneu	28328	18825	8052	6305	42966	40931
14	recursion	8318	7143	5096	N/A	N/A	N/A
15	statemate	2486	3810	1260	1120	6207	5898

N/A = Not applicable.
buffer = Because of the buffer limitation in the test board, it is not possible to measure the WCETs.

Furthermore, we observed that:

- The synthesized code from C-Lab, the car window lift control (`statemate`) and the Petri-Net (`nsichneu`), contains mainly control structure: switch statements, conditionals, both deeply nested. In executables, large switch statements may lead to indirect jumps that use registers to reference contents of switch tables. Resolving these jumps precisely requires knowledge about register contents, as provided by value analysis. Here aiT's value analysis helps resolve these jumps, and discover infeasible paths, which is the reason why aiT's tightness is better on these benchmarks.
- Chronos did not implement value analysis therefore its overapproximation on these two benchmarks is higher than average.

8 Discussion

We summarize the main findings of the paper.

8.1 Tool Assessment with respect to Industrial Requirements

Consisting of commercial and research prototypes from five different countries, the participating WCET tools represent the state of the art in the domain. This allows us to evaluate the Challenge results and in this way draw a picture of the state of the art in WCET, from the perspective of industrial expectations.

Tool Support. Fundamentally, a WCET tool has to support the target platform and the program to be analyzed, possibly aided by human effort.

Table 13 WCET Tightness: Prediction vs. Measurement

Nr.	Benchmarks	aiT C16x	aiT ARM7	aiT MPC565	Chronos SimpleScalar (SIO)	Chronos SimpleScalar (CIO)	Chronos SimpleScalar (COO)
1	adpcm	N/A	N/A	N/A	65.07%	89.48%	151.35%
2	cnt	3.2 %	1.19%	1.95%	2.17%	15.25%	53.66%
3	compress	37.58%	1.55%	38.64%	0.24%	289.33%	500.48%
4	cover	3.69%	0.03%	16.45%	N/A	N/A	N/A
5	crc	N/A	N/A	N/A	110.62%	130.26%	194.37%
6	duff	4.05%	0.04%	31.81%	N/A	N/A	N/A
7	edn	10.54%	2.72%	N/A	2.24%	4.26%	41.33%
8	insertsorts	3.18%	0.05%	3.84%	0.45%	13.56%	31.19%
9	janne_complex	2.7%	0.24%	6.69%	2.16%	76.21%	121.63%
10	matmult	2.15%	N/A	N/A	0.0%	3.05%	31.59%
11	ndes	12.97%	2.06%	N/A	1.61%	24.18%	60.22%
12	ns	2.68%	5.39%	N/A	24.66%	32.02%	81.35%
13	nsichneu	5.34%	0.01%	3.42%	115.84%	127.87%	138.27%
14	recursion	21.13%	4.31%	8.46%	N/A	N/A	N/A
15	statemate	5.39%	0.05%	2.7%	79.2%	160.75%	132.02%
1-15	Average	8.8%	1.5%	12.7%	33.7%	80.5%	131.5%

N/A = Not applicable.

The target processors the PowerPC, ARM (ARM7 / ARM9), and C16X are the top three popular processors in industry, according to a survey in 2004 on industrial requirements [44].

The benchmark selection covers tasks from both the automotive and the avionics domain with hard timing constraints, in this way, testing scalability and applicability to real-life code. Academic benchmarks provide extreme test cases in terms of determination of loop bounds and infeasible paths. The benchmarks also represent different coding styles ranging from code automatically generated from models to hand-written code.

All benchmark programs could be analyzed, albeit some required annotation. However, not all tools can solve all benchmarks. In general, coding style and language constructs have a strong influence on analyzability of the program, and required human effort. For instance, using for-loops instead of while-loops makes the detection of loop bounds easier for WCET tools.

Certain types of control flow such as "unstructured code" were not supported by Bound-T and Chronos. aiT and SWEET could obtain control flow automatically. This was difficult for Bound-T and Chronos. SWEET is a special case because it extracts information from the source code unlike the other tools.

While many loop bounds could be detected automatically, certain types of loop conditions and nesting were difficult. More precisely, the following types of loop conditions caused trouble: loop conditions depending on complex boolean operations, which may happen in model-based code as, e.g., generated by Statemate, or loop conditions that use a comparison of floating point numbers, e.g. in numerical computations. Nested loops with complex dependencies between iteration variables are hard in general.

In terms of loop bounds, aiT only required annotations in case of the complex loop conditions and nesting mentioned above; all other bounds were detected automatically.

SWEET could analyze all of its own benchmark programs automatically, i.e., annotations were only used to improve tightness. However, SWEET failed on the other benchmarks because its special compiler could not produce the intermediate code needed by SWEET.

Concerning Bound-T, some difficulties in control-flow reconstruction could be resolved by annotation, except for the unsupported types of control flow. Loop bound detection was less automatic than in case of aiT and SWEET. It posed the largest challenge for Bound-T, causing high analysis times and timeouts.

Chronos had problems with control-flow reconstruction. Loop bound detection, in general, required more annotations than aiT. However, Chronos exclusively succeeded together with SWEET in `janne_complex`.

Correctness and Precision. We conducted measurements to assess the correctness and precision of static timing analysis results from aiT and Chronos. The tools always produced a WCET value above the measured execution times, i.e. no obvious bugs were revealed. By comparison of the WCET analysis results with and without annotations, we observed that, annotations improved the precision of WCET results. Furthermore by comparison of the WCET analysis results with annotations to the observed measured execution time, we observed that, the overestimation averaged over all benchmarks in case of aiT was in the range of 1.5% - 12.7% for the different targets, and for Chronos 33.7% - 131.5%.

Usability Assessment. We wanted to compare the pay-off of invested effort in terms analysis success and precision. We allocated an approximately equal amount of time and effort on each tool.

Properties of the analysis engine immediately affect usability, such as the degree of automation. In this context, degree of automation, support for certain language constructs, usability of tools, and precision of computed WCET bounds are important evaluation criteria. A one-click solution is always desirable, however none of the tools achieves this in all cases. In this paragraph, we focus on the human effort needed to achieve reasonable results and the assistance from tools in achieving this goal. In case user effort is required to improve analysis results, the user should receive assistance from the tool in providing necessary annotations. aiT points to program locations in the source code where annotations such as loop bounds are necessary, and it warns if program annotations seem unreasonable.

Our experience is that a unified GUI for project management, as present in aiT, avoids switching software interfaces and saves time and effort.

Integration requirements. Integration into a software development process requires that the tool supports the employed compiler. All tools except for SWEET[3] only need the executable and not the source code of a program. This is useful, since, in industry, source code is not always available for WCET analysis.

It is desirable that the output format follow a standard like XML to make processing by scheduling tools easier, as is done by aiT.

Analysis time. We give some information about analysis time, although this was not a central point in the test. For the benchmarks, we observe that most tools ran in tolerable time. aiT and SWEET took a few second to analyze each of the benchmark programs, even without manual annotations. Results for Chronos were obtained with the commercial ILP solver CPLEX, all other tools used the academic ILP solver `lp_solve` [33]. Chronos running time with CPLEX was a few seconds. In some cases, Bound-T timed out if no annotation was given.

8.2 Achievements and Limitations

Achievements. To our knowledge, this is the first work in which the major WCET tools have been evaluated by an independent party.

Publications typically focus on novel analysis techniques. In the Challenge, tools could excel by solving small, but hard benchmark programs. On the other hand,

---

[3] On the other hand, SWEET explores what is possible with source-code-based analysis and showed promising results in automatic flow analysis.

to be useful in the large, some investment into infrastructure is needed. Often, this work by itself does not yield publishable results. Our tool comparison encourages both such effort and novel analysis techniques.

In the course of the evaluation, we discovered bugs, some of which led to wrong results. These bugs were reported to tool developers and were subsequently fixed.

Limitations. The submitted tools perform WCET analysis for different target processors. As a consequence, different compilers and thus different executables were analyzed by the respective tools. Hence it is not possible to compare the absolute WCET values.

It would be interesting to compare different approaches in the two analysis phases of WCET tools, i.e., the program analysis and the microarchitectural analysis. These are however implemented in different tools and intermediate results are not always accessible.

9 Conclusion

The entered tools have shown their strengths in different aspects of WCET analysis: Both commercial tools aiT and Bound-T were able to handle the two fly-by-wire PapaBench test programs. SWEET automatically analyzed 88% of the benchmark test programs. The analysis time of Chronos was very short when using CPLEX. aiT was able to handle every kind of benchmark and every test program that was tested in the Challenge. aiT is able to support WCET analysis even for complex processors. aiT provided a user-friendly WCET analysis environment. aiT was able to demonstrate the precision of their WCET analysis results. aiT demonstrates its leading position through all its features, which contributes to its position as an industry-strength tool satisfying the requirements from industry as posed by EADS Airbus and proven by the accomplishment in various projects.

The external test trips gave the tool developers immediate feedback on tool usability and possible software faults. This feedback has been taken into account by the developers. All developers have provided their best support and cooperation for the external test.

The Challenge has encouraged WCET research and activities in the WCET community. During the Challenge, many developers were engaged in developing and improving their tools further. A total of 25 updated versions of the software were submitted for the external tests. Bugs were fixed, and their latest releases in the Challenge have demonstrated the improvements of the WCET tools.

The Challenge improved the awareness of the tool developers concerning requirements from industry, e.g. support for model-based design. At the ISoLA 2006 conference, the Challenge has also caught the attention of more WCET developers. They showed their interest to participate in future WCET Challenges. We anticipate

positive effects of the Challenge on the WCET community in tool development and industrial application.

Future work. It would be desirable to compare tools supporting the same target in the Challenge. The impact of different hardware features on precision and analyzability could be incorporated into a future Challenge. Furthermore, the benchmarks could be partitioned into different categories, e.g. industrial application code vs. WCET research code, model-based synthesized code vs. hand-written code. Current WCET analysis tools do not exploit the specifics of synthesized code. Directions for future research could include a study about code generated by different model-based development environments, like SCADE [24], Matlab/Simulink [20], and AS-CET [4]. For example, it might be worthwhile to investigate if control-flow dependencies arising from the structure of state automata can be leveraged to improve tightness of WCET estimates.

References

1. aiT. http://www.absint.de/ait/.
2. Altium Tasking Compiler. http://www.altium.com/TASKING/.
3. ARM9. http://www.arm.com/products/CPUs/families/ARM9Family.html.
4. ASCET. http://www.etas.com/de/products/ascet_software_products.php.
5. Bound-T. http://www.tidorum.fi/bound-t/.
6. C-Lab. http://www.c-lab.de/.
7. Chronos. http://www.comp.nus.edu.sg/~rpembed/chronos/.
8. Cosmic. http://www.cosmicsoftware.com/.
9. CPLEX. http://www.ilog.com/products/cplex/.
10. Freescale MPC565. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC565.
11. Gaisler BCC Compiler. http://gaisler.com/doc/bcc.pdf.
12. GNU C Compiler. http://gcc.gnu.org/.
13. Graphviz. http://www.graphviz.org/.
14. H8/300. http://eu.renesas.com/fmwk.jsp?cnt=h8300_series_landing.jsp&fp=/products/mpumcu/h8_family/h8300_series/.
15. HCS12. http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=02Wcbf8WD69BXm.
16. IAR. http://www.iar.com/.
17. Infineon. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b2f42075b.
18. ISO9241. http://en.wikipedia.org/wiki/ISO_9241.
19. Mälardalen Benchmarks. http://www.mrtc.mdh.se/projects/wcet/sweet.html.
20. Matlab/Simulink. http://www.mathworks.com.
21. MIPS. http://www.mips.com/.
22. PapaBench. http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97/.
23. Paparazzi Project. http://paparazzi.enac.fr/wiki/index.php/Main_Page.
24. SCADE. http://www.esterel-technologies.com/products/scade-suite/.
25. SimpleScalar. http://www.simplescalar.com/.
26. SPARCV7/V8. http://www.sparc.org/specificationsDownload.html.
27. SPEC95. http://www.spec.org/cpu95/.
28. SWEET. http://www.mrtc.mdh.se/projects/wcet/sweet.html.
29. TMS470. http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=203&familyId=454.
30. WCET Discussion Group. http://tech.groups.yahoo.com/group/wcet/.
31. WindRiver Compiler. http://www.windriver.com/products/development_suite/wind_river_compiler/.
32. WCET Tool Challenge 2006. Internet, 2006. http://www.idt.mdh.se/personal/jgn/challenge/.
33. M. Berkelaar. lp solve: A mixed integer linear program solver. Tech. rep., Eindhoven University of Technology, 1997.
34. Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University.
35. Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD Thesis, Universität des Saarlandes, 1997. http://rw4.cs.uni-sb.de/~ferdi/publications.html.
36. Jan Gustafsson. The WCET Tool Challenge 2006. In Bernhard Steffen Tiziana Margaris, Anna Philippeu, editor, *Second International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 233–240, November 2007.
37. David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
38. Seoul National University Real-Time Research Group. SNU-RT. http://realtime.snu.ac.kr/realtime/.
39. University of Maryland. The Omega project. http://www.cs.umd.edu/projects/omega/.

40. Lili Tan. The Worst-Case Execution Time Tool Challenge 2006: The External Test. In Bernhard Steffen Tiziana Margaris, Anna Philippeu, editor, *Second International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 241–248, November 2007.

41. Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-Based Worst-Case Execution Time Analysis. In *SEUS*, pages 7–10. IEEE Computer Society, 2005.

42. Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter Puschner. Automatic timing model generation by cfg partitioning and model checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.

43. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

44. Reinhard Wilhelm, Jakob Engblom, Stephan Thesing, and David B. Whalley. Industrial Requirements for WCET Tools - Answers to the ARTIST Questionnaire. In Jan Gustafsson, editor, *WCET*, volume MDH-MRTC-116/2003-1-SE, pages 39–43. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 2003.