WCET Challenge 2006 – technical report

Jan Gustafsson, Mälardalen University, Västerås, Sweden

Abstract—**Worst Case Execution Time (WCET) analysis has a growing importance for real-time systems, to guarantee correct timing, and to be an aid in developing such systems. The WCET tools are currently making their way out to the market, and there are many research groups active in developing prototype tools using new and better ways of calculating estimates or bounds on the WCET.**

The purpose of the WCET Tool Challenge is to be able to study, compare and discuss the properties of different WCET tools and approaches, to define common metrics, and to enhance the existing WCET benchmarks. The WCET Tool Challenge has been designed to find a good balance between openness for a wide range of analysis approaches, and specific participation guidelines to provide a level playing field. This should make results transparent and facilitate friendly competition among the participants.

This report describes the participating tools as well as the results of the Challenge 2006. There is also an accompanying report by Lili Tan on the external tests of the tools.

The WCET Tool Challenge is intended to be an annual event.

I. Introduction

There have been WCET benchmarks around for some years. The idea of a WCET Tool Challenge came up during the spring of 2006, inspired by other areas where competitions or similar have been used to be able to discuss and compare different approaches (e.g., in automated reasoning (The CADE ATP System Competition) [1].

The WCET Tool Challenge has been performed during the autumn of 2006. It has concentrated on three aspects of WCET analysis:

1. flow analysis,
2. required user interaction,
3. performance.

Two commercial tools, aiT [2] and Bound-T [3], and three research prototype tools, MTime [4], SWEET [5], and Chronos [6], participated. An external research assistant and the development teams made the actual tests of the tools. The tests targeted on a set of benchmark programs. For more details on the setup of the event, consult the Challenge web page [7]. The Challenge has been supported by the Compilers and Timing Analysis Cluster of ARTIST2 [8].

There are certainly more WCET tools being developed that could have participated in the Challenge. To the author's knowledge we have OTAWA [9], RapiTime [10], SYMTA/P [11], and Heptane [12]. For a number of reasons, the developers of these tools did not participate.

There are two main of reasons for not joining the Challenge:

- The time schedule is too tight or the timing is not OK.
- The Challenge is still too biased towards static analysis methods and there is no support for measurement based methods.

For the next Challenge these problems should be solved.

A. Goals

The goals of the WCET Tool Challenge are the following:

1) To exhibit the wide range of timing analysis tools available today:

- using static program analysis, or
- combining analysis and measurements,
- for various target processors,
- in various application domains,
- supporting various programming languages and design tools,
- academic, commercial; free or at a charge.

2) To illuminate the features, abilities and intended uses of each tool:

- in finding the feasible execution paths in the SW,
- in modelling complex processor and system HW,
- in deriving useful WCET bounds or estimates,
- in usability, scalability and adaptability,
- in the range of supported targets (processors, compilers, ..)

3) To collect and maintain a growing set of community standard benchmark programs and related test suites that:

- contain typical (both easy and hard) programming constructs,
- can be analyzed by several tools with comparable results,
- test enough of the actual behaviour of each benchmark to satisfy measurement-based tools and to validate results from static-analysis tools, and ideally, have known exact answers (paths and WCETs).

B. Aspects of WCET analysis

1) Area 1 - Flow analysis

The purpose of the flow analysis phase is to extract the dynamic behaviour of the program. This includes information on which functions get called, loop bounds, if there are dependencies between if-statements, etc. We use the following flow analysis metrics:

- number of automatically found loop bounds (including context-depending bounds, like triangular loop limits and loops in functions called from several sites)

- tightness of these (compared to real loop bounds, assuming they are known)
- the number of automatically found infeasible paths
- their reduction effects on the WCET estimates
- the number of automatically found correct memory accesses
- the number of automatically found resolved call targets (function pointers)

2) Area 2 - Required user interaction

This area of evaluation is concerned with the amount of work that is involved with setting up a WCET calculation to receive a result. One important metric for this area is number of program-specific manual annotations. Necessary annotations (like CPU type, frequency etc) can be excluded from the number.

3) Area 3 – Performance

This area is about the bottom line: the final WCET value and the performance of the tool.

We use the following metrics:

- estimated WCET value (in clock tics and µs)
- tightness of the estimated WCET value (assuming the real WCET is known)
- limits of program sizes to be handled
- analysis time and memory requirements for the analysis

C. WCET tool rounds

Aspects may be non-orthogonal and influence each other. For example, much preparation work may give a better (tighter) WCET. This is expected and normal, and shows the signs of a flexible WCET tool. The same tool can be used for different aspects with different setups. Therefore we suggest that each tool is used in three rounds for each target processor:
1) One initial round with no manual annotations for loop bounds etc. Necessary annotations (like CPU type, frequency etc) are however allowed. This round requires loop bounds to be found automatically; otherwise the analysis will not give a WCET bound at all for benchmarks containing loops.
2) A basic round with the smallest set of manual annotations possible to get a WCET bound.
3) An optimal round with the largest set of annotations to get as tight WCET bound as possible.

The required user interaction is of course growing for each round. For each round, the metrics are measured for the three aspects. For each metric, the complete setup is described.

D. Carrying the evaluation out

There have been two possibilities:
1) An external evaluation, carried out by Lili Tan (lili.tan@icb.uni-due.de), who is a research assistant in the research group of Dependability of Computing Systems at the University of Duisburg-Essen. This has the advantage of letting an external person try out the tools and give an independent feedback of the usability of the tools, without bias of any WCET tool developer.
2) The evaluation is carried out by the developer or supplier of the tool.

E. Selection of benchmark programs, processors and compilers.

The benchmarks will represent different types of codes, for example code with different types of loops, infeasible paths, automatically generated code, hardware specialized code, and also large real world programs. A mix of single-path programs and multi-path programs will be included.

We have used open source benchmark programs from the Mälardalen WCET benchmark [5] and PapaBench [14]. These benchmarks are available on the web.

Each participant selects up to three processors for which to do the analyses; for example one simple (e.g., Renesas H8), one medium complex (e.g., ARM7/9, C167NEC, V850E) and one very complex (e.g., PowerPC), if possible. As there is no overview over which compilers are supported by which tools, we let the participants decide on one or two compiler(s).

II. The Benchmarks

The Mälardalen Benchmarks

These benchmarks are collected from several different research groups and tool vendors around the world and are available on the web. The programs contain different types of code constructs (loops, nested loops, arrays, matrixes, bit operations, recursion, unstructured code, floating point etc.). The benchmarks are single path programs when run as they are provided, but can be run or analysed in multi path mode using the provided annotations, which define multiple values to certain variables at certain program points.

The benchmark programs have varying sizes (from ≈ 100 to 1300 lines of lines of source code). Each benchmark is provided as a C source file. In total there is 30+ benchmark programs, of these 15 were selected for the Challenge. The selection was made so that as many different types of code was included as possible. Also, no floating-point programs were included, because many of the processors to be analysed by the tools do not support this in hardware.

Table I on the next page describes the Mälardalen Benchmarks.

Legend:
I = uses include files.
E = calls external library routines.
S = always single path program (no potential flow depenency on external variables).
L = contains loops.
N = contains nested loops,
A = uses arrays and/or matrixes.
B = uses bit operations.
R = contains recursion.
U = contains unstructured code.
F = uses floating point calculation.
LOC = lines of source code.

Additional comments to the benchmarks concerning WCET analysis problems:
adpcm: This program is a signal processing application; the comments call it an implementation of the Adaptive Differential Pulse Code Modulation algorithm. The program was originally written with floating point computation. For the WCET benchmark, this was changed (by the WCET Challenge Steering Group) into (nonsensical) integer computation. This means that some computations overflow when compiled with 16-bit integers, as was the case for the H8/300 target, which constitutes a problem for the WCET analysis.
crc: The main function calls *icrc* twice with an initialization included in the first call but not in the second call. This constitutes a problem for the WCET analysis.
duff: This program creates a control flow graph that is not reducible, because the loop has multiple entry points (multiple loop heads).
insertsort: The maximum number of iterations in the inner loop depends on the counter (i) of the outer loop ("triangular loop" problem).
nsichneu: This program has an enormous number (10^{75}) of potential paths.
recursion: The program computes Fibonacci numbers using a recursive function. This constitutes a problem for the WCET analysis. (Due to a mistake on the benchmark web page, the mutual recursion included in the code was commented away in the program).

TABLE I
MÄLARDALEN BENCHMARKS

Program	**Description**	**Comment**	**I**	**E**	**S**	**L**	**N**	**A**	**B**	**R**	**U**	**F**	**LOC**
adpcm	Adaptive pulse code modulation algorithm.	Completely well-structured code.				√							87
cnt	Counts non-negative numbers in a matrix	Nested loops, well-structured code. Simple code.				√	√	√					267
compress	Data compression program.	Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer containing totally random data.				√	√	√					508
cover	Program for testing many paths.	A loop containing many switch cases.			√	√							240
crc	Cyclic redundancy check computation on 40 bytes of data.	Complex loops, lots of decisions, loop bounds depend on function arguments, function that executes differently the first time it is called.			√	√		√	√				128
duff	Using "Duff's device" from the Jargon file to copy 43 byte array.	Unstructured loop with known bound, switch statement.			√	√					√		86
edn	Finite Impulse Response (FIR) filter calculations.	A lot of vector multiplications and array handling.			√	√	√	√	√				285
insertsort	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of n2/2 iterations (triangular loop).				√	√	√					92
janne_complex	Nested loop program.	The inner loops number of iterations depends on the outer loops current iteration number.			√	√	√						64
matmult	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.			√	√	√	√					163
ndes	Complex embedded code.	A lot of bit manipulation, shifts, array and matrix calculations.				√		√	√				231
ns	Search in a multi-dimensional array.	Return from the middle of a loop nest, deep loop nesting (4 levels).				√	√	√					535
nsichneu	Simulate an extended Petri Net.	Automatically generated code containing large amounts of if-statements (more than 250).				√							4253
recursion	A simple example of recursive code.	Both self-recursion and mutual recursion are used.			√					√			41

PapaBench

This benchmark is based on a real and complete real-time embedded application, developed in the Paparazzi project, to be used on Unmanned Aerial Vehicles (UAV).

The Paparazzi UAV contains two ATMEL AVR processors and thus the PapaBench benchmark contains two programs, one for each processor. Each program is in turn divided into a number of 'tasks'. Most tasks are executed cyclically with a fixed period. Some tasks are event-triggered. There is no kernel. The tasks are executed by an eternal loop in the *main()* function. There are also interrupt handlers.

The benchmark has a total size of ≈ 2 000 lines of source code. The software constitutes quite complex code that makes extensive use of floating point computation and mathematical library routines.

The part *papa_ap* is the "autopilot" part of the PapaBench program for an autonomous aircraft. It executes a flight plan and communicates with the "fly-by-wire" part papa_fbw.

The part *papa_fbw* of the PapaBench program is the "fly-by-wire" program for an autonomous aircraft. It communicates with the "autopilot" part (program *papa_ap*) and runs servo loops to control the aircraft.

It is difficult to find the number of loops in the source code because C macros are used extensively to create syntactical structures including loops.

There were some problems with the PapaBench software. Some changes were made during the Challenge, for example a hard-coded memory address for the "dummy" in/output of AVR was changed to a static int array.

III. The Tools

A. *Commercial tools*

There were two commercial tools in the Challenge; aiT and Bound-T. The descriptions below are fetched from [14] and are slightly shortened since some details are left out.

1) *AiT*

The purpose of AbsInt's timing-analysis tool aiT is to obtain upper bounds for the execution times of code snippets (e.g. given as subroutines) in executables. These code snippets may be tasks called by a scheduler in some real-time application, where each task has a specified deadline. aiT works on executables because the source code does not contain information on register usage and on instruction and data addresses. Such addresses are important for cache analysis and the timing of memory accesses in case there are several memory areas with different timing behavior.

Apart from the executable, aiT might need user input to be able to compute a result or to improve the precision of the result. User annotations may be written into parameter files and refer to program points by absolute addresses, addresses relative to routine entries, or structural descriptions (like the first loop in a routine).

Alternatively, they can be embedded into the source code as special comments. In that case, they are mapped to binary addresses using the line information in the executable.

Apart from the usual user annotations (loop bounds, flow facts), aiT supports annotations specifying the values of registers and variables. The latter is useful for analyzing software running in several different modes that are distinguished by the value of a mode variable.

The aiT versions for all supported processors share a common architecture:

—First, the control flow is reconstructed from the given object code by a bottom up approach. This annotated control-flow graph serves as the input for the following analysis steps.

—Next, value analysis computes ranges for the values in the processor registers at every program point. Its results are used for loop bound analysis, for the detection of infeasible paths depending on static data, and to determine possible addresses of indirect memory accesses.

—aiT's cache analysis relies on the addresses of memory accesses as found by value analysis and classifies memory references as sure hits and potential misses.

—Pipeline analysis predicts the behavior of the task on the processor pipeline.

The result is an upper bound for the execution time of each basic block in each distinguished execution context.

—Finally, bound calculation (called path analysis in the aiT framework) determines a worst-case execution path of the task from the timing information for the basic blocks.

The structuring of the whole task of determining upper bounds into several phases allows different methods tailored to the subtasks to be used. In aiT's case, value analysis and cache/pipeline analysis are realized by abstract interpretation, a semantics-based method for static program analysis.

Path analysis is implemented by integer linear programming. Reconstruction of the control flow is performed by a bottom-up analysis. Detailed information about the upper bounds, the path on which it was computed, and the possible cache and pipeline states at any program point are attached to the call graph / control-flow graph and can be visualized in AbsInt's graph browser aiSee.

Limitations of the Tool. aiT includes automatic analysis to determine the targets of indirect calls and branches and to determine upper bounds of the iterations of loops. These analyses do not work in all cases. If they fail, the user has to provide annotations.

2) *Bound-T*

The tool determines an upper bound on the execution time of a subroutine, including called functions. The input is a binary executable program with (usually) an embedded symbol table (debug information). The tool is able to compute upper bounds on some counter-based loops. For other loops the user provides annotations, called assertions in Bound-T. Annotations can also be given for variable values to support the automatic loop-bounding.

The output is a text file listing the upper bounds etc. and graph files showing call-graphs and control-flow graphs for display with the DOT tool [15].

Reading and decoding instructions is hand-coded based on processor manuals. The processor model is also manually constructed for each processor. Bound-T has general facilities for modelling control flow and integer arithmetic, but not for modelling complex processor states. Some special-purpose static analyses have been implemented, for example for the SPARC register-file overflow and underflow traps and for the concurrent operation of the SPARC Integer Unit and Floating Point Unit. Both examples use (simple) abstract interpretation followed by ILP (Integer Linear Programming).

The control-flow graph (CFG) is often defined to model the processor's instruction- sequencing behaviour, not just the values of the program counter. A CFG node typically represents a certain pipeline state, so the CFG is really a pipeline-state graph. Instruction interactions (e.g. data-path blocking) are modelled in the time assigned to CFG edges.

Counter-based loops are bounded by modelling the task's loop-counter arithmetic. The computational effect of each instruction is modelled as a relation between the "before" and "after" values of the variables (registers and other storage locations). The relation is expressed in Presburger Arithmetic as a set of affine (linear plus constant term) equations and inequalities, possibly conditional. Instruction sequences are modelled by concatenating (joining) the relations of individual instructions. Branching control-flow is modelled by adding the branch condition to the relation. Merging control-flow is modelled by taking the union of the inflowing relations.

To bound loop iterations, Bound-T first re-analyses the model of the loop body in more detail to find loop-counter variables. A loop counter is a loop-variant variable such that one execution of the loop body changes the variable by an amount that is bounded to a finite interval that does not contain zero. If Bound-T also finds bounds on the initial and final values of the variable, a simple computation gives a bound on the number of loop iterations.

Bound-T uses the Omega Calculator from Maryland University [16] to create and analyze the equation set. Loop-bounds can be context-dependent if they depend on scalar pass-by-value parameters for which actual values are provided at the top (caller end) of a call-path.

The worst-case path and the upper bound for one subroutine are found by the Implicit Path Enumeration Technique, applied to the control-flow graph of the subroutine. The *lp_solve* tool is used [17]. If the subroutine has context-dependent loop bounds, the IPET solution is computed separately for each context (call path).

Annotations are written in a separate text file. The program element to which an annotation refers is identified by a symbolic name (subroutine, variable) or by structural properties (loops, calls). The structural properties include nesting of loops, location of calls with respect to loops, and location of variable reads and writes.

Limitations of the Tool. The task to be analyzed must not be recursive. The control-flow graphs must be reducible. Dynamic (indexed) calls are only analyzed in special cases, when Bound-T's data-flow analysis finds a unique target address.

Dynamic (indexed) jumps are analyzed based on the code patterns that the supported compilers generate for switch/case structures, but not all such structures are supported.

Bound-T can detect some infeasible paths as a side-effect of its loop-bound analysis. There is, however, no systematic search for such paths. Points-to analysis (aliasing analysis) is weak, which is a risk for the correctness of the loop-bound analysis.

The bounds of an inner loop cannot depend on the index of the outer loop(s).

For such "non-rectangular" loops Bound-T can often produce a "rectangular" upper bound. Loop-bound analysis does not cover the operations of multiplication (except by a constant), division or the logical bit-wise operations (and, or, shift, rotate).

The task to be analyzed must use the standard calling conventions. Furthermore, function pointers are not supported in general, although some special cases such as statically assigned interrupt vectors can be analyzed.

No cache analysis is yet implemented (the current target processors have no cache or very small and special caches). Any timing anomalies in the target processor must be taken into account in the execution time that is assigned to each basic block in the CFG. However, the currently supported, cacheless processors probably have no timing anomalies. As Bound-T has no general formalism (beyond the CFG) for modelling processor state, it has no general limitations in that regard, but models for complex processors would be correspondingly harder to implement in Bound-T.

B. Prototype research tools

1) MTime

The Vienna MTime analysis tool aims at assessing the timing behavior of automatically generated program code for embedded systems.

The first step performed by the method is a static program analysis.

Next, the control-flow graph is partitioned automatically into custom-sized program segments in order to reduce the complexity. Then, for all paths spawning across these program segments test data is generated in order to execute exactly the desired paths. After this, the execution times of these paths are measured on the actual target hardware. Finally, the results of these measurements are safely composed into a final WCET bound by using integer linear programming.

The key challenges when implementing the prototype have been the completely automatic program segmentation, test data generation (using model checking) and predictable execution time measurements.

The current limitations of the proof-of-concept prototype implementation are the missing support of function calls (because when using code generators functions can be inlined) and generic loops.

The key strengths of the method are that it can be applied without any user interaction, the tool is modular, hardware timing behavior is contributed by predictable execution measurements performed on the actual target hardware and the WCET bound can be calculated using well established calculation methods.

2) SWEET

SWEET has been developed in a modular fashion, allowing for different analyses and tool parts to work rather independently. The tool architecture conforms to the general scheme for WCET analysis, consisting of three major phases: a flow analysis, a processor-behavior analysis and an estimate calculation. The analyses communicate through two well-defined data structures, the scope graph with flow facts [18], (representing the result of the flow analysis), and the timing model [19], (representing the result of the processor-behavior analysis). In essence, SWEET offers the following functionality:

—Automatic flow analysis on the intermediate code level.

—Integration of flow analysis and a research compiler.

—Connection between flow analysis and processor-behavior analysis.

—Instruction cache analysis for level one caches.

—Pipeline analysis for medium-complexity RISC processors.

—A variety of methods to determine upper bounds based on the results of flow- and pipeline analysis.

Unlike most WCET analysis tools, SWEET's flow analysis is integrated with a research compiler. The flow analysis is performed on the intermediate code (IC) of the compiler, after structural optimizations. Thus, the control structure of the IC and the ob ject code is similar, and the flow analysis results for the IC are valid for the ob ject code as well.

SWEET's flow analysis is based on a multi-phase approach. A program slicing is used to restrict the flow analysis to only those parts of the program that may affect the program flow. The analysis is handled by the abstract execution, a form of symbolic execution based on abstract interpretation. The analysis uses abstract interpretation to derive safe bounds on variables values at different points in the program. However, rather than using traditional fixed-point iteration, loops are "rolled out" dynamically and each iteration is analysed individually in a fashion similar to symbolic execution. The abstract execution is able to automatically calculate both loop bounds and infeasible path information.

SWEET's processor-behavior analysis is highly decoupled from the flow analysis, and based on a two-phase approach. In the first phase, the memory access analysis, memory areas accessed by different instructions are determined. The result of the analysis is a set of "execution facts" which are used in the pipeline analysis. Such facts specify the memory area(s) that an instruction may reference or if the instruction may hit and/or miss the cache. Execution facts can also specify other factors like assumptions on branch prediction outcomes, and the precise set available depends on the target processor.

The pipeline analysis is performed by simulating object code sequences through a trace-driven cycle-accurate CPU model. The pipeline analysis has been explicitly designed to allow standard CPU simulators to be used as CPU models. However, this requires that the simulator is clock-cycle accurate, can be forced to perform its simulation according to given instruction sequences and corresponding execution facts, and does not suffer from timing anomalies.

Consecutive simulation runs starting with the same basic block in the code are combined to find timing effects across sequences of two or more blocks in the code [19]. The analysis assumes that there is a known upper bound on the length of block sequences that can exhibit timing effects; this value can be greater than two even on quite simple processors.

SWEET's estimate calculation phase support three different type of calculation techniques, all taking the same two data structures as input. A fast path-based technique, a global IPET technique and a hybrid clustered technique. The clustered calculation can perform both local IPET and/or local path-based calculations (the decision on what to use is based on the flow information available for the specific program part under analysis).

SWEET uses DOT from GraphViz [15] to graphically visualize its results.

Limitations of the Tool. Each part of the tool, flow analysis, processor-behavior analysis, and bound calculation have their individual limitations. The flow analysis can handle ANSI-C programs including pointers, unstructured code, and recursion. However, to make use of the automatic flow analysis the program must be compiled with the research compiler that SWEET is integrated with, otherwise flow facts must be manually given. There are also some limitations inherent to the used research compiler, e.g. the use of dynamically allocated memory is currently not supported, and annotations will be needed in such cases.

The current memory access analysis does not handle data caches. Only one-level instruction caches are supported. The pipelines that are amenable to SWEET's pipeline analysis are limited to in-order pipelines with bounded long-timing effects and no timing anomalies. In particular, out-of-order pipelines are not handled.

The path-based bound calculation requires that the code of the task is well-structured. The IPET-based and clustered calculation methods can handle arbitrary task graphs.

3) Chronos

Chronos is an open-source static WCET analysis tool. The purpose of Chronos is to determine a tight upper bound for the execution times of a task running on a modern processor with complex micro-architectural features. The input to Chronos is a task written in C and the configuration of the target processor. The frontend of the tool performs data flow analysis to compute loop bounds. If it fails to obtain certain loop bounds, user annotations have to be provided. The user may also input infeasible-path information to improve the accuracy of the results. The frontend maps this information from the source code to the binary executable.

The core of the analyzer works on the binary executable. It disassembles the executable to generate the control flow graph (CFG) and performs processor-behavior analysis on this CFG. Chronos supports the analysis of (i) out-of-order pipelines, (ii) various dynamic branch prediction schemes, (iii) instruction caches, and the interaction among these different features to compute a tight upper bound on the execution times.

The core of the analyzer determines upper bounds of execution times of each basic block under various execution contexts such as correctly predicted or mispredicted jump of the preceding basic blocks and cache hits/misses within the basic block. Determining these bounds is challenging for out-of-order processor pipelines due to the presence of timing anomalies. It requires the costly enumeration of all possible schedules of instructions within a basic block. Chronos avoids this enumeration via a fixed-point analysis of the time intervals (instead of concrete time instances) at which the instructions enter/leave different pipeline stages.

Next, the analyzer employs Integer Linear Programming (ILP) to bound the number of executions corresponding to each context. This is achieved by bounding the number of branch mispredictions and instruction cache misses. Here ILP is used to accurately model branch prediction, instruction cache as well as their interaction. The analysis of branch prediction is generic and parameterizable w.r.t. the commonly used dynamic branch prediction schemes including GAg and gshare. Instruction caches are analyzed using the ILP-based technique proposed by Li, Malik, and Wolfe. However, the integration of cache and branch prediction requires analyzing the constructive and destructive timing effects due to cache blocks being "prefetched" along the mispredicted paths. This complex interaction is accounted for in the analyzer by suitably extending the instruction cache analysis.

Finally, bounds calculation is implemented by the IPET technique by converting the loop bounds and user provided infeasible-path information to linear flow constraints.

Limitations of the Tool. Chronos currently does not analyze data caches. Since the focus is mainly on processor-behavior analysis, the tool performs limited data flow analysis to compute loop bounds. The tool also requires user feedback for infeasible program paths.

IV. Results

A. Results from the developers

Since the MTime tool does not support function calls, and all benchmarks contain such calls, no results from MTime were available this time. All other four tools have reported their results. We give an individual summary for each of the tools.

We regret that the result tables for the four tools are different. The reason for this is that no common reporting format was enforced during the Challenge. This should be changed in coming Challenge rounds.

1) aiT

The report from aiT was short, a summary of the results fitted into two Excel sheets. Tables II to IV show the content of the Excel sheet. As we can see, aiT succeeded in analyzing all of the benchmark programs for all three target processors (C16x, ARM TMS470, and PowerPC MPC565). They also measured the execution times for the Mälardalen benchmarks (except for the programs marked with "buffer", which means that the buffer of their measurement equipment was too small to handle the large runtime of the program). The overestimation done by aiT is in most cases below 10%. The analysis for ARM shows especially low overestimation. No measurements were provided for the PapaBench benchmark.

MinAn is the minimal number of annotations needed to get a WCET-analysis result (loop bounds, recursion bounds, flow annotations, ...). Not counted are annotations for the runtime libraries of the used compilers, which are usually delivered together with the analyser.

AT is the runtime of the analysis in seconds for the set of optimal annotations. The analysis used the following hardware: PC with Pentium 4, 2.x-3.x Ghz, 2 GB RAM, as much swap-space as needed. In the time values, fractions of seconds are removed, as they are not reproducible. Not all tests are run on exactly the same machine. The *matmult* test for ARM7 needs more than 2GB RAM for the set of the best annotations, therefore it is run on a 64bit 3.2 Ghz Xeon with 8 GB RAM, with normal set of annotations it runs on the other normal machines in 3 seconds. In table IV, A-Loops are number of loop bounds found automatically without any annotations, counted over loop contexts.

Analysis problems encountered for the aiT tool
No real problems have been reported. Obviously, not all loop bounds were found automatically, so some had to provided manually (see the colunmn "MinAn").

TABLE II
RESULTS FOR AIT – MÄLARDALEN BENCHMARKS

wcet_bench	**C16x**			**ARM**			**PowerPC**		
	C16x	**cycles**		**TMS470**	**cycles**		**MPC565**	**cycles**	
Program	**Measured**	**WCET**	**Over-estimation**	**Measured**	**WCET**	**Over-estimation**	**Measured**	**WCET**	**Over-estimation**
adpcm	buffer	558 342	-	buffer	1 375 886	-	buffer	430 274	-
cnt	19622	20 250	3,20%	16 853	17 053	1,19%	7 235	7 376	1,95%
compress	27308	37 570	37,58%	19 970	20 280	1,55%	6 824	9 461	38,64%
cover	10080	10 452	3,69%	6 778	6 780	0,03%	4 299	5 006	16,45%
crc	buffer	275 910	-	buffer	196 007	-	buffer	98 830	-
duff	6916	7 196	4,05%	4 610	4 612	0,04%	1 028	1 355	31,81%
edn	838686	927 068	10,54%	299 734	307 889	2,72%	buffer	88 381	-
insertsort	4720	4 870	3,18%	3 990	3 992	0,05%	1 770	1 838	3,84%
janne_complex	1294	1 330	2,78%	827	829	0,24%	359	383	6,69%
matmult	936602	956 710	2,15%	438 435	448 261	2,24%	buffer	237 736	-
ndes	401294	453 348	12,97%	190 530	194 448	2,06%	buffer	130 025	-
ns	73738	75 712	2,68%	36 097	38 043	5,39%	buffer	18 215	-
nsichneu	28328	29 840	5,34%	18 825	18 827	0,01%	8 052	8 327	3,42%
recursion	8318	10 068	21,04%	7 143	7 451	4,31%	5 096	5 527	8,46%
statemate	2486	2 620	5,39%	3 810	3 812	0,05%	1 260	1 294	2,70%

TABLE III
RESULTS FOR AIT – PAPABENCH

papa_bench		**C16x**		**ARM**		**PowerPC**	
		C16x	**cycles**	**TMS470**	**cycles**	**MPC565**	**cycles**
Task	**Executable**		**WCET**		**WCET**		**WCET**
T1 / _test_ppm_task	fly_by_wire		16 604		9 875		1 242
T2 / _send_data_to_autopilot_task	fly_by_wire		4 758		3 197		331
T3 / _check_mega128_values_task	fly_by_wire		9 512		4 092		437
T4 / _servo_transmit	fly_by_wire		2 390		1 909		1 249
T5 / _check_failsafe_task	fly_by_wire		9 444		4 058		432
T6 / _radio_control_task	autopilot		20 516		15 972		2 247
T7 / _stabilisation_task	autopilot		9 936		4 239		340
T8 / _link_fbw_send	autopilot		170		144		81
T9 / _receive_gps_data_task	autopilot		33 942		23 618		2 764
T10 / _navigation_task	autopilot		163 266		87 979		5 303
T11 / _altitude_control_task	autopilot		2 134		915		95
T12 / _climb_control_task	autopilot		8 458		4 129		247
T13 / _reporting_task	autopilot		8 286		11 172		4 465

TABLE IV
NUMBER OF FOUND LOOP BOUNDS AND ANALYSIS TIMES FOR AIT

wcet_bench	**C16x**			**ARM**			**PowerPC**		
	C16x			**TMS470**			**MPC565**		
Program	**A-Loops**	**MinAn**	**AT**	**A-Loops**	**MinAn**	**AT**	**A-Loops**	**MinAn**	**AT**
adpcm	28	3	50	24	3	3	25	2	23
cnt	6	0	18	6	0	4	6	0	2
compress	4	7	35	3	6	4	2	6	3
cover	3	0	52	3	0	1	3	0	7
crc	4	1	219	3	1	2	5	0	128
duff	0	2	3	0	2	1	1	1	1
edn	20	5	143	26	0	14	18	0	61
insertsort	1	1	5	2	1	2	1	1	1
janne_complex	0	2	3	0	1	2	0	2	1
matmult	13	0	66	11	0	743	11	0	301
ndes	41	0	406	12	0	3	12	0	54
ns	9	0	40	6	0	1	6	0	6
nsichneu	0	1	58	1	0	30	1	1	8
recursion	0	1	1	0	1	1	0	1	1
statemate	0	1	13	0	1	15	0	1	53

papa_bench	**C16x**			**ARM**			**PowerPC**		
	C16x			**TMS470**			**MPC565**		
Task	**A-Loops**	**MinAn**	**AT**	**A-Loops**	**MinAn**	**AT**	**A-Loops**	**MinAn**	**AT**
Executable: fly_by_wire									
T1 /_test_ppm_task	0	0	28	0	0	2	0	0	1
T2 /_send_data_to_autopilot_task	1	0	7	1	0	2	1	0	1
T3 /_check_mega128_values_task	0	0	9	0	0	2	0	0	1
T4 /_servo_transmit	1	0	1	1	0	1	1	0	1
T5 /_check_failsafe_task	0	0	9	0	0	1	0	0	1
Executable: autopilot									
T6 /_radio_control_task	0	0	27	6	0	4	0	0	4
T7 /_stabilisation_task	0	0	13	0	0	3	0	0	1
T8 /_link_fbw_send	0	0	1	0	0	1	0	0	1
T9 /_receive_gps_data_task	0	4	38	2	4	6	0	4	2
T10 /_navigation_task	0	12	208	15	12	41	0	14	14
T11 /_altitude_control_task	0	0	3	0	0	1	0	0	2
T12 /_climb_control_task	0	0	9	0	0	4	0	0	1
T13 /_reporting_task	0	0	22	0	0	6	0	0	1

2) Bound-T

The report from Tidorum Ltd. [20] is very well-written and ambitious. The Bound-T tool analysed binaries generated by two compilers (GNU gcc compiler and the Gaisler Research Bare C Compiler) for two targets; Renesas H8/300 and SPARC V7/V8. The analyses were performed using the Round 1–3 approach described in the setup of the Challenge. Actually, the Bound-T tests are the only to report this three-round approach. No measurements were provided.

The host computer was a Compaq Presario X1000 laptop with an Intel Centrino processor running at 1.4 GHz and 512 MB RAM. The operating system was Debian Linux.

The Bound-T report thoroughly describes the used processors and compilers. We omit these descriptions here for space reason and refer to the report. In the table below, the results for Bound-T are described. The table is an excerpt from the report. For example, we only give results for the round that gives a WCET value (round 1 if all loop bounds were found automatically, else round 2). We also omit programs where Bound-T failed.

The columns have the following meanings:

– Comp: The name of the compiler. The remaining columns in the row report results from the executable generated with this compiler. If this column is blank, the column Loops in the row report properties of the source code and the remaining columns are unused. The compiler also identifies the target processor as follows:

gcc: The GNU compiler for the H8/300.

bcc: The Gaisler Research Bare C Compiler (based on GCC) for the SPARC.

– Loops: The number of loops in the program. The number of loops in the executable is often larger than in the source code because the compiler may generate loops (eg. for C expressions like n << k) and there may be loops in library routines. For the SPARC/BCC target the loops in the irreducible library routines are not included.

– Bound: The number of loops for which Bound-T found iteration bounds. Note that some loops may be counted twice, if Bound-T finds both context-independent and context-dependent bounds for the same loop.

– Ass: The number of loops for which iteration bounds was asserted, perhaps for each Round as in the preceding column, or for which other assertions (such as variable value bounds) were used that let Bound-T find iteration bounds. Blank for the source code row.

– AT: Analysis time, in seconds of real (wall-clock) time. The time is measured with one user logged in but no other heavy activity. From run to run the time varies about 5 - 10%.

Analysis problems encountered for the Bound-T tool

The benchmark *duff* is irreducible and cannot be analysed by Bound-T. The loop termination logic in *janne_complex* is too complex for Bound-T and too complex for manual reasoning. The recursive program *recursion* cannot be analysed by Bound-T.

The analysis during round 1 (trying to find loop bounds automatically) took too long for 7 benchmarks and had to be aborted. In addition, the analysis failed with an error in the Omega Calculator in a further 3 benchmarks. For these programs, loop assertions had to be calculated manually.

TABLE V
RESULTS FOR BOUND-T

Program	Comp	Loops	Bound	Ass	AT	WCET
adpcm		20				
	gcc	31	26	5	9.5	2 002 692
	bcc	18	15	3	2.3	803 089
cnt		4				
	gcc	5	5		0.5	45 806
	bcc	4	4		0.4	19 628
compress		8				
	gcc	15	4	11	2.6	586 093
	bcc	8	4	4	31.5	122 249
cover		3				
	gcc	3	3		6.6	7 742
	bcc	3	3		1.0	3173
crc		3				
	gcc	3	3		1.3	164 118
	bcc	3	2	1	0.5	57 663
edn		13				
	gcc	23	19	4	136.9	1 514 112
	bcc	15	12	3	1.3	426 779
insertsort		2				
	gcc	2	1	1	0.3	7 760
	bcc	2	1	1	0.2	2078
matmult		5				
	gcc	6	6		0.8	1506520
	bcc	5	5		0.4	615345
ndes		12				
	gcc	21	19		39.0	823 416
	bcc	12	12		1.3	82676
ns		4				
	gcc	4	4		38.1	20976
	bcc	4	4		3.0	7097
nsichneu		1				
	gcc	1	0	1	215.6	104 522
	bcc	1	0	1	6.8	20 756
papa_ap		?				
	gcc	85	23	54	1 224.5	20 266 762
	bcc	45	2	1	1.0	62 753
papa_fbw		?				
	gcc	25	9	15	6.5	1 150 867
	bcc	6	3	1	1.0	9008

3) *MTime*

Since the MTime tool does not support function calls, and all benchmarks contain such calls, no results from MTime were available this time.

4) *SWEET*

The tables VI to VIII show the results for SWEET and the Mälardalen WCET benchmarks. For the analyses we used a PC running Windows XP with Intel processor, clock frequency 1.06 GHz, and 1.49 Gb of primary memory. The WCET is expressed in clock cycles for ARM9. Please note that the machine model of ARM9 has not been verified to 100%. Since there are no measurements, no figures of overestimation can be given. Analysis time (AT) is given in seconds (excluding loading of files). Times in seconds; have been rounded to one decimal in the table. The number of loops is the number found in the analysed intermediate code.

Table VI. Single path analysis, basic loop bound calculation.

This analysis uses no annotations. SWEET calculates all loop bounds automatically.

Table VII. Single path analysis, advanced loop bound calculation plus infeasible path calculation.

For these analyses we add analysis of infeasible paths, and show the number of such paths found. For some programs, this can give tighter WCET estimates. In this table, “% longer” means extra time used for the analysis compared to the analysis time in Table VI. The column “% better” shows the difference (relative reduction) of the calculated WCET compared to Table V.

Table VIII. Multi path analysis, basic loop bound calculation.

For some of the files, the analysis was “forced” to analyse several paths (multi path analysis). This was accomplished by defining annotations that assigned multiple values to some variables at some program point. The annotation files (with comments) are available at the Mälardalen benchmark web page.

4. Table IX. Multi path analysis, advanced loop bound calculation plus infeasible path calculation.

The same settings for as Table VII above are used, but for multi path analysis.

Analysis problems encountered for the SWEET tool

We had overestimation of one loop in *adpcm* probably due to pessimistic handling of overflow. The PapaBench programs could not be analysed by SWEET because of problems with finding and analyzing the C-files that corresponds to the used library routines for floating point calculations.

5) *Chronos*

Chronos was run for three processor configurations (simple in-order, complex in-order and complex out-of-order). Table X presents the results. For each of the benchmarks, the data representing these three configurations is presented on an own line.

Analysis problems encountered for the Chronos tool

The programs *cover* and *duff* are not analyzable using Chronos because they contain switch/case statements that are compiled into address tables and register-indirect jumps. The program recursion is not analyzable using Chronos since the benchmark contains recursive function call. The program autopilot in PapaBench is not analyzable using Chronos since this benchmark contains unstructured code (goto jumps). These types of code are not handled in the current tool release.

Reasons for some of the large overestimations are given in the Chronos report. They originate probably from overestimations of loop bounds (especially nested loops) and missing information on infeasible paths. Another reason for the seemingly large overestimations is that the simulations may use data that do not lead to the worst-case behavior.

For the benchmark *fly_by_wire* in PapaBench, compilation and estimation should be done from the command line instead of the Chronos GUI. This benchmark is not executable by Chronos, so there are no simulated values in this case.

TABLE VI
RESULTS FOR SWEET – SINGLE PATH BASIC

Program	**#Loops**	**AT flow analysis**	**AT low level analysis**	**AT total**	**WCET**
adpcm	27	73.9	10.9	84.8	2 165 650
cnt	4	1.3	0.2	1.5	36 719
compress	11	2.7	1.6	4.3	206 480
cover	3	3.8	7.0	10.8	73 128
crc	6	8.7	0.6	9.3	834 159
duff	2	0.4	0.2	0.6	5 525
edn	12	5.3	1.2	6.5	1 425 085
insertsort	2	0.6	0.1	0.7	31 163
janne_complex	2	0.1	0.0	0.1	12 039
matmult	7	11.8	0.3	12.1	2 532 706
ndes	12	31.6	2.6	34.3	795 425
ns	4	4.4	0.1	4.5	130 733
nsichneu	1	41.6	48.3	89.9	119 707
recursion	0	0.8	0.1	0.9	29 079
statemate	1	5.6	8.9	10.2	15 964

TABLE VII
RESULTS FOR SWEET – SINGLE PATH ADVANCED

Program	**AT flow analysis**	**AT low level analysis**	**AT total**	**% longer**	**WCET**	**#infeasible paths found**	**% better**
adpcm	75.3	9.7	84.9	0	2 162 122	182	0
cnt	1.3	0.2	1.5	3	35 319	10	4
cnt	2.7	1.5	4.2	-2	49 896	45	76
cover	3.8	10.3	14.1	31	33 485	576	54
crc	9.0	0.7	9.7	4	830 278	36	0
duff	0.5	0.2	0.6	13	4 720	54	15
edn	5.5	1.2	6.8	4	1 425 085	17	0
insertsort	0.6	0.1	0.7	3	18 167	3	42
janne_complex	0.1	0.0	0.1	27	2 523	10	79
matmult	12.4	0.3	12.8	6	2 532 706	12	0
ndes	32.2	2.9	35.1	2	793 905	132	0
ns	4.5	0.1	4.6	1	130 671	7	0
nsichneu	41.5	27.9	69.4	-23	57 247	877	52
recursion	0.8	0.1	0.9	11	20 033	16	31
statemate	1.2	5.6	6.9	32	8 451	328	47

TABLE VIII
RESULTS FOR SWEET – MULTI PATH BASIC

Program	**AT flow analysis**	**AT low level analysis**	**AT total**	**WCET**
crc	40.1	0.6	40.7	834 159
edn	30.8	1.2	32.0	1425 085
insertsort	31.5	0.1	31.5	31 163
janne_complex	10.7	0.0	10.7	19 104
ns	70.5	0.1	70.7	130 733
nsichneu	53.5	48.2	101.8	119 707

TABLE IX
RESULTS FOR SWEET – MULTI PATH ADVANCED

Program	**AT flow analysis**	**AT low level analysis**	**AT total**	**% longer**	**WCET**	**#infeasible paths found**	**% better**
crc	41.6	0.7	42.3	4	834 088	36	0
edn	32.2	1.2	33.4	4	1 425 085	17	0
insertsort	31.8	0.1	31.9	1	18 167	3	42
janne_complex	11.1	0.1	11.2	4	3 154	10	83
ns	69.4	0.1	69.5	-2	130 671	7	0
nsichneu	54.1	33.2	87.4	-14	41 303	877	65

TABLE X
RESULTS FOR CHRONOS

Program	#Loops	# Loops found automatically	ILP formulation time	ILP solution time	WCET estimated	WCET simulated	Overestima-tion
adpcm	18	4	0.273	0.019	265 588	160 891	65%
			1.256	1.021	347 742	183 526	89%
			1.035	0.556	317 354	126 258	151%
cnt	4	4	0.029	0.012	4 896	4 792	2%
			0.084	0.020	6 438	5 586	15%
			0.067	0.020	5 401	3 515	54%
compress	8	1	0.231	0.015	5 873	5 859	0%
			0.618	0.054	29 215	7 504	289%
			0.451	0.055	28 487	4 744	500%
crc	3	1	0.098	0.013	47 786	22 688	111%
			0.338	0.052	61 849	26 861	130%
			0.267	0.064	53 275	18 098	194%
edn	12	12	0.058	0.025	89 401	87 444	2%
			0.187	0.086	113 612	108 973	4%
			0.147	0.088	89 030	62 995	41%
insertsort	2	1	0.018	0.001	901	897	0%
			0.045	0.036	1 549	1 364	14%
			0.036	0.033	1245	949	31%
*janne_*	2	2	0.098	0.017	189	185	2%
			0.321	0.039	800	454	76%
			0.028	0.248	789	356	122%
matmult	5	5	0.025	0.018	186 903	186 899	0%
			0.063	0.048	191 615	185 937	3%
			0.054	0.054	119 526	90 834	32%
ndes	12	7	0.083	0.029	66 655	65 600	2%
			0.414	1.022	107 589	86 639	24%
			0.317	1.186	5 918	53 625	60%
ns	5	0	0.027	0.021	8 199	6 577	25%
			0.071	0.033	9 991	7 568	32%
			0.032	0.058	8 676	4 784	81%
nsichneu	1	1	9.791	0.447	13 609	6 305	116%
			31.562	3.461	97 908	42 966	128%
			22.939	3.356	97 525	40 931	138%
statemate	1	0	2.494	0.137	2 007	1 120	79%
			7.546	0.832	16 185	6 207	161%
			5.708	0.813	16 103	5 898	173%
fly_by_wire			0.254	0.019	7 712	-	-
			0.638	0.670	11 980	-	-
			0.503	0.807	9 983	-	-

B. *Overview of results.*

Table XI shows the success rate expressed as #WCET results/#programs. No results from MTime.

Some tools have problems with some of the following: function calls, unstructured code, recursion, handling library files, handling large programs (run-time of tool), specifying correct loop bounds, and annotating infeasible paths.

TABLE XI
ANALYSIS SUCCESS RATES

Tool	**Successes, Mälardalen benchmarks**	**Successes, PapaBench**
aiT	15/15	2/2
Bound-T	11/15	2/2
SWEET	15/15	0/2
Chronos	12/15	1/2

Table XII shows the success of automatic loop bound calculation expressed as #Loop bounds found/(total #loops analysed in successfully analysed programs). No results from MTime. The "-" sign in the aiT row means that the total number of loops were not provided by aiT.

There was no indication of the precision in the found loop bounds.

TABLE XI
LOOP BOUND RESULTS

Tool	**Successes, Mälardalen benchmarks**	**Successes, PapaBench**
aiT	129/- (C16x) 91/- (PPC)	2/- (C16x) 2/- (PPC)
Bound-T	90/114 (H8/300) 62/75 (SPARC)	32/110 (H8/300) 5/51 (SPARC)
SWEET	94/94	0/0
Chronos	38/72	0/1

Only SWEET reports results from infeasible path analysis. The other tools require infeasible paths to be found and set manually.

For usability aspects, see the external report by Lili Tan [14].

V. CONCLUSIONS

This first Challenge has been performed successfully in spite of some initial unclearness, debate and delay. We can note in the tables from the different tests that there is no common format of the results, and that different developers have made different tests sometimes. We still consider the Challenge to be meaningful and worthwhile, and we can draw the following conclusions:

- The tests have been a real challenge to the participating WCET tools. We have a success range from 0% to 100% in terms of how many of the benchmark programs that were analyzable by a certain tool.
- The tests have clearly pointed out problems existing in the tools as well as in the benchmarks and the used compilers.
- Most of the tools find more than half of the loop bounds automatically. Only one tool finds infeasible paths automatically.
- Several bugs in both the tools and the benchmarks have been corrected during the Challenge.
- Actual WCET estimates cannot be compared this time since the developers support different processors and compilers.
- The quality of WCET estimates is hard to judge for all tools but aiT, since aiT was the only tool to provide measurements for some of the benchmarks. Chronos provided simulated values that indicate the possible size of overestimation.

ACKNOWLEDGMENT

This work is being supported by ARTIST2 European Network of Excellence.

We would also like to thank the other members of the WCET Challenge Working Team:

Reinhard Wilhelm <wilhelm@cs.uni-sb.de>, Reinhard v. Hanxleden <rvh@informatik.uni-kiel.de>, Steffen Goerzig <steffen.goerzig@daimlerchrysler.com>, Paul Levi Paul.Levi@informatik.uni-stuttgart.de, and Klaus Echtle <klaus.echtle@icb.uni-due.de>.

REFERENCES

[1] http://www.cs.miami.edu/~tptp/CASC
[2] absInt. aiT tool homepage, 2006. www.absint.com/ait.
[3] Tidorum. Bound-T tool homepage, 2006. www.tidorum.fi/bound-t.
[4] Vienna real-time systems group homepage, 2005. www.vmars.tuwien.ac.at.
[5] Mälardalen University. WCET project homepage, 2006. www.mrtc.mdh.se/projects/wcet.
[6] The Chronos WCET analysis tool homepage, 2006. www.comp.nus.edu.sg/~rpembed/chronos.
[7] http://www.idt.mdh.se/personal/jgn/challenge
[8] **http://www.artist-embedded.org/FP6/intranet/ClusterPages/CompilersTA/**
[9] OTAWA homepage: http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=28
[10] RapiTime WCET tool homepage, 2006. www.rapitasystems.com.
[11] *Cost-efficient worst-case execution time analysis in industrial practice*, Jan Staschulat, Jörn C. Braam, Rolf Ernst, Thomas Rambow, Rainer Schlör, and Rainer Busch. In Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISoLA'06), November 2006.
[12] Homepage for the Heptane WCET analysis tool, 2006. www.irisa.fr/aces/work/heptane-demo/heptane.html.
[13] Homepage for PapaBench, 2007. http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97

[14] Reinhard Wilhelm et al. *The Worst-Case Execution Time Problem --- Overview of Methods and Survey of Tools*. Submitted to ACM Transactions on Programming Languages and Systems 2006.

[15] Graphviz homepage 2007. http://www.graphviz.org/

[16] *An Exact Method for Analysis of Value-based Array Data Dependences*, William Pugh, David Wonnacott. In Proceedings of the Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing 1993.

[17] http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml

[18] Ermedahl, A. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Ph.D. thesis, Uppsala University, Uppsala, Sweden, 2003.

[19] Engblom, J. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* Ph.D. thesis, Uppsala University, Uppsala, Sweden, 2002.

[20] Tidorum Ltd. Doc.ref. TR-RP-2006-009, issue 1, October 6, 2006.

[21] Tan, L. *The Worst Case Execution Time Tool Challenge 2006: The External Test.* To be included in the Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISoLA'06), November 2006.