THE MÄLARDALEN WCET BENCHMARKS: PAST, PRESENT AND FUTURE

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper
School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden.

{jan.gustafsson,adam.betts,andreas.ermedahl,bjorn.lisper}@mdh.se

Abstract

Modelling of real-time systems requires accurate and tight estimates of the Worst-Case Execution Time (WCET) of each task scheduled to run. In the past two decades, two main paradigms have emerged within the field of WCET analysis: static analysis and hybrid measurement-based analysis. These techniques have been succesfully implemented in prototype and commercial toolsets. Yet, comparison among the WCET estimates derived by such tools remains somewhat elusive as it requires a common set of benchmarks which serve a multitude of needs.

The Mälardalen WCET research group maintains a large number of WCET benchmark programs for this purpose. This paper describes properties of the existing benchmarks, including their relative strengths and weaknesses. We propose extensions to the benchmarks which will allow any type of WCET tool evaluate its results against other state-of-the-art tools, thus setting a high standard for future research and development.

We also propose an organization supporting the future work with the benchmarks. We suggest to form a committee with a responsibility for the benchmarks, and that the benchmark web site is transformed to an open wiki, with possibility for the WCET community to easily update the benchmarks.

1. Introduction

Bounding the Worst-Case Execution Time (WCET) of real-time software is crucial when developing and verifying real-time systems. These bounds must be safe and tight (i.e., as close to the actual WCET as possible).

WCET analysis attempts to deliver such a bound. Its techniques can broadly be categorised as follows:

- *End-to-end measurements* is the traditional approach and is used widely in industry. Test-vector generation algorithms attempt to stress the longest execution time of the program under analysis. To try and bypass any optimism, some additional margin is added to the longest recorded time and this is considered as the WCET estimate.

- *Static analysis* relies on mathematical models of the software and hardware involved. The hardware model allows the execution time of individual instructions to be gleaned. The software model represents possible execution flows. Combining these models with information about the maximum number of times loops are iterated, which paths through the program that are feasible, execution frequencies of code parts, etc., results in a WCET estimate. Provided that the models are correct, the WCET estimate is always safe, i.e., greater than or equal to the actual WCET.

- *Hybrid measurement-based analysis* operates similarly to static analysis, except it does not create a hardware model. Rather, it uses measurements to derive execution times of small program parts, before combining them using flow information in the WCET calculation. Although the WCET estimate is generally more accurate than that computed by static analysis, there is a possibility of underestimation if testing has not sufficiently stressed the execution times of the small program parts.

A number of WCET analysis tools have emerged in recent years. Academic toolsets of note include: OTAWA [14], Chronos [11], SWEET [12], and Heptane [8]. Some of the developed techniques have also migrated into fully-fledged commercial tools, including: RapiTime [18], aiT [1], and Bound-T [21]. However, a comparison between these tools, and the associated methods and algorithms, requires a common set of benchmarks. The typical evaluation metric is the accuracy of the WCET estimate, but of equal importance are other properties such as performance (i.e., scalability of the approach) and general applicability (i.e., ability to handle all code constructs found in real-time systems). In summary, it is very useful to have an easily available, thoroughly tested, and well documented common set of benchmarks in order to enable comparative evaluations of different algorithms, methods, and tools.

The Mälardalen WCET benchmarks have been assembled with the above goals in mind. This paper describes properties of the existing benchmarks, including their relative strengths and weaknesses. In particular, we propose to extend the benchmarks with new types of codes, which will raise the standard for future research and development of WCET algorithms, methods, and tools.

We also propose an organization supporting the future work with the benchmarks.

The rest of the paper is organized as follows: Section 2 places the Mälardalen WCET benchmarks into context by reviewing other benchmark suites on offer. Following that, Section 3 describes the WCET benchmarks and Section 4 evaluates them, presenting ideas for development of an extended version. Section 5 concludes the paper and presents future work.

2. Related Benchmarks

Benchmarking is a problem not only in the WCET community but across various computing disciplines. For this reason, the number of available benchmark suites for computer science is large. A typical goal of these benchmarks is to evaluate performance for various computing areas, for example for integer and floating-point calculations, e.g., the Drystone benchmark [3], and for stressing a system's processor, memory subsystem and compiler, e.g., the SPEC CPU2006 benchmark [2].

As an exhaustive examination of benchmarking suites for computer science is beyond the scope of this paper, this section instead relates to those which have most relevance to WCET analyses.

The goal of the EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [4] is to specify benchmarks for both the hardware and software utilised in embedded systems. At the time of this writing, eight suites are available, each of which is designed to stress a particular type of workload in the embedded domain, including: automotive, digital imaging, digital entertainment, energy consumption, mobile Java applications, networking, office automation (e.g. printers), and telecommunications. All of the benchmarks are written in C or Java. A benefit of the EEMBC benchmarks is that they are continually maintained and updated, as the consortium is run as a non-profit organisation. However, the downside is that gaining access to the benchmarks requires a licence, even for academics.

Drawing motivation from this deficiency, the MiBench benchmarks [13] were proposed, which are open source in comparison and are written in C. Similarly to the EEMBC suites, MiBench splits its programs into six distinct groups: automotive, consumer, networking, office automation, security, and telecommunications. Given their strong correlation to the embedded domain, many of these benchmarks appear to be suitable candidates for WCET analysis, although they have been sparsely used in the WCET community.

In the WCET tool challenge of 2008 [22], several other benchmarks were introduced. The DEBIE-1 benchmark is a satellite application, written in C, consisting of six tasks. The main appeal of this benchmark is that it is a realistic application, having been initially supplied by Space Systems Finland Ltd, and it is shipped with a test harness (developed by Tidorum Ltd [21]) thereby easing measurement-based analyses. The other four benchmarks used, *rathijit_1* through *rathijit_4*, were provided by Saarland University, and aim to have large instruction cache and data cache footprints. The DEBIE-1 benchmark is not open source, but can be requested from Tidorum Ltd, whereas the *rathijit* applications are freely available.

PapaBench [15] is another recently proposed benchmark in the WCET community, which is based on an actual real-time application from within the avionic industry. PapaBench is a real-time embedded benchmark derivated from the software of a GNU-license UAV, called Paparazzi. Formerly driving a bi-processor AVR architecture, the application C sources have been adapted to compile under several other platforms. Similarly to the DEBIE-1 benchmark, PapaBench consists of a number of tasks and interrupts.

3. The Mälardalen WCET Benchmarks

The Mälardalen WCET benchmarks were collected in 2005 from several researchers within the WCET field. Properties of each benchmark program (which are all written in C) are listed in Table 1 and 2.

The purpose of the Mälardalen WCET benchmarks is to have a common, easily available, set of test programs for WCET methods and tools. The benchmarks includes a broad set of program constructs to support testing and evaluation of WCET tools.

The Mälardalen WCET benchmarks are available on a web page [23]. The benchmark programs are marked with the following properties: I = uses include files (i.e., uses more than one file), E = calls

Program	**Description**	**Comments**
`adpcm`	Adaptive pulse code modulation algorithm.	Completely well-structured code.
`bs`	Binary search for the array of 15 integer elements.	Completely structured.
`bsort100`	Bubblesort program.	Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.
`cnt`	Counts non-negative numbers in a matrix.	Nested loops, well-structured code.
`compress`	Compression using lzw.	Adopted from SPEC95 for WCET-calculation. Only compression is done on a buffer (small one) containing totally random data.
`cover`	Program for testing many paths.	A loop containing many switch cases.
`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.
`duff`	Using "Duff's device" to copy 43 byte array.	Unstructured loop with known bound, switch statement
`edn`	Finite Impulse Response (FIR) filter calculations.	A lot of vector multiplications and array handling.
`expint`	Series expansion for computing an exponential integral function	Inner loop that only runs once, structural WCET estimate gives heavy overestimate.
`fdct`	Fast Discrete Cosine Transform.	A lot of calculations based on integer array elements.
`fft1`	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.	A lot of calculations based on floating point array elements.
`fibcall`	Iterative Fibonacci, used to calculate fib(30).	Parameter-dependent function, single-nested loop
`fir`	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.	Inner loop with varying number of iterations, loop-iteration dependent decisions.
`insertsort`	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $(n^2)/2$ iterations (triangular loop).

Table 1. Benchmark programs (part 1)

external library routines, S = is a single path program (no flow dependency on external variables), L = contains loops, N = contains nested loops, A = uses arrays and/or matrices, B = uses bit operations, R = contains recursion, U = contains unstructured code, and F = uses floating point calculation. The size of source code file (bytes), as well as LOC = number of lines of source code, is also provided.

There are some main categories of benchmark programs:

- Well-structured code (all benchmark programs except `duff`)
- Unstructured code (`duff`)
- Array and matrix calculations (`bs, bsort100, edn, fdct, fft1, insertsort, ludcmp, matmult, minver, ndes, ns, qsort-exam, qurt, select, st`)
- Nested loops (`adpcm, bsort100, cnt, compress, crc, edn, expint, fft1, fibcall, fir, insertsort, janne_complex, ludcmp, matmult, minver, ns, qsort-exam, select`)
- Input dependent loops (`bsort100, janne_complex, insertsort`)
- Inner loops depending on outer loops (`crc, fir, janne_complex, insertsort`)
- Switch cases (`cover`)
- Nested if-statements (`nsichneu`)
- Floating point calculations (`fft1, lms, ludcmp, minver, qsort-exam, qurt,`

Program	**Description**	**Comments**
`janne_complex`	Nested loop program.	The inner loops number of iterations depends on the outer loops current iteration number.
`jfdctint`	Discrete-cosine transformation on 8x8 pixel block.	Long calculation sequences (i.e., long basic blocks), single-nested loops.
`lcdnum`	Read ten values, output half to LCD.	Loop with iteration-dependent flow.
`lms`	LMS adaptive signal enhancement. The input signal is a sine wave with added white noise.	A lot of floating point calculations.
`ludcmp`	LU decomposition algorithm.	A lot of calculations based on floating point arrays with the size of 50 elements.
`matmult`	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.
`minver`	Inversion of floating point matrix.	Floating value calculations in 3x3 matrix. Nested loops (3 levels).
`ndes`	Complex embedded code. A lot of bit manipulation, shifts, array and matrix calculations.	A lot of bit manipulation, shifts, array and matrix calculations.
`ns`	Search in a multi-dimensional array.	Return from the middle of a loop nest, deep loop nesting (4 levels).
`nsichneu`	Simulate an extended Petri net.	Automatically generated code with more than 250 `if`-statements.
`qsort-exam`	Non-recursive version of quick sort algorithm.	The program sorts 20 floating point numbers in an array. Loop nesting of 3 levels.
`qurt`	Root computation of quadratic equations.	The real and imaginary parts of the solution are stored in arrays.
`recursion`	A simple example of recursive code.	Both self-recursion and mutual recursion are used.
`select`	A function to select the Nth largest number in a floating point array.	A lot of floating value array calculations, loop nesting (3 levels).
`sqrt`	Square root function implemented by Taylor series..	Simple numerical calculation.
`st`	Statistics program.	This program computes for two arrays of numbers the sum, the mean, the variance, and standard deviation, and the correlation coefficient between the two arrays.
`statemate`	Automatically generated code.	Generated by the STAtechart Real-time-Code generator STARC.

Table 2. Benchmark programs (part 2)

`select, sqrt, st`)

- Bit manipulation (`crc, edn, fdct, lcdnum, ndes`)
- Recursive code (`recursion`)
- Automatically generated code (`nsichneu, statemate`)

3.1. Additional information provided

The web page also includes meta-data for the benchmarks: inputs for some of the benchmarks, number of loop iterations, and some types graphs. This is described in more detail in the following.

Single-path/multi-path benchmarks and inputs to the benchmarks. The programs in the benchmark can all be run ”as is”, i.e., the programs contain their own inputs. This means that they execute a single path. However, most realistic programs are run with different inputs at different invocations.

If the inputs can affect the control ow, the programs WCET is usually highly dependent on inputs.

For WCET analysis, it is important to know the possible values of the input variables since these, in general, must be constrained as much as possible in order to obtain tight program flow constraints from the flow analysis. For an embedded program or task (written in C or a similar language), the input variables can be:

- Values read from the environment using primitives such as ports or memory mapped I/O,
- Parameters to `main()` or the particular function that invokes the task, and
- Data used for keeping the state of tasks between invocations or used for task communication, such as external variables, global variables or message queues.

Therefore, we have defined multiple input values for some of the benchmarks, to be able to test and evaluate such input dependency. These inputs are provided as intervals, i.e., limits to the inputs. The inputs are stored on the web page as ”input annotations” (.ann files) in SWEET format.

Loop bounds. Each benchmark program has been run either ”as is” (in single mode), or, if inputs are defined, with all inputs. The loop bounds that have been found are stored in a file at the web site. The loop bounds for the program are either exact (in the single mode case) or the maximum possible with the possible inputs, as defined on the web site. This information can be useful when doing loop bound analysis.

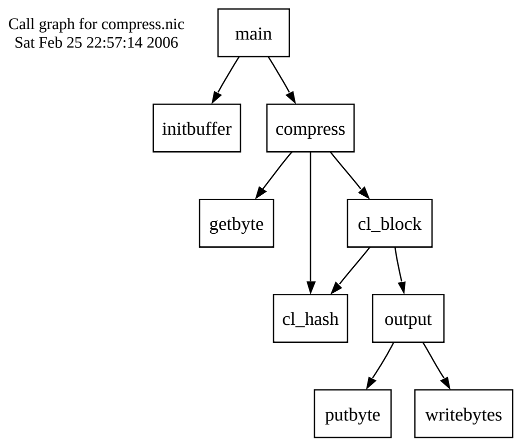

Figure 1. Example of a call graph for a benchmark program (compress).

Call graph and scope hierarchy graph. The web site contains some graphs generated by the SWEET tool. For each benchmark file, a *call graph* (see Figure 1) is provided as a PDF file. A *scope hierarchy graph* is also available (see Figure 2), which is a context sensitive graph showing calls to functions and entries to loops. The root scope (at the top) is typically the main function, or the top function in a subgraph. The (iteration) scope is either a function or a loop, and constitutes a (possibly) looping entity in the program. The arrow from one scope to another below represents a call in the case of a function scope, or a loop invocation in the case of a loop scope. If loops are considered as a special case of (tail recursive) functions (which is a common way to look at loops), the call graph becomes the scope hierarch graph. The scope hierarchy graph is context sensitive, which means that each call site to a function creates a unique scope of the called function. The graph gives a possibility to find all, possibly looping, scopes (functions and loops) in the program.

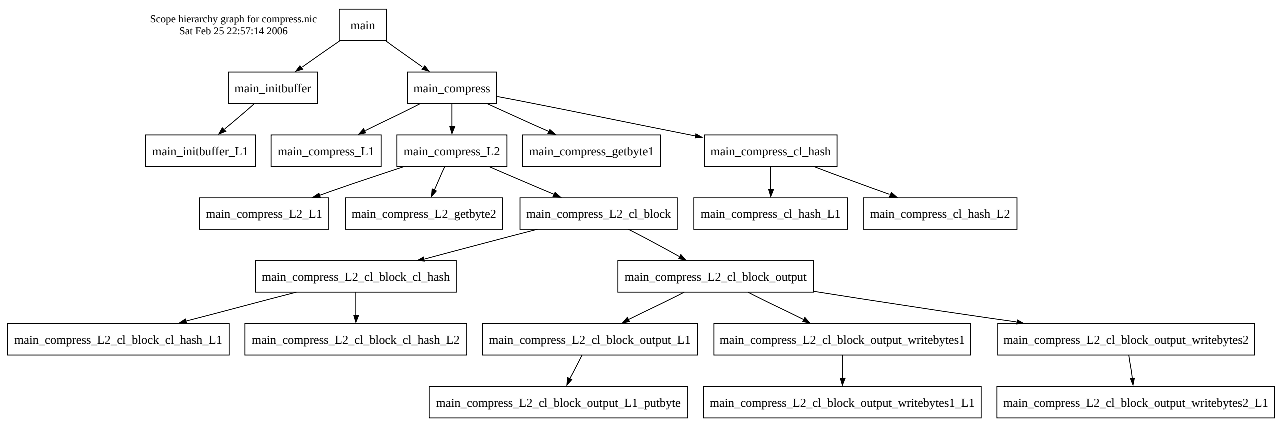
Scope hierarchy graph for compress.nic
Sat Feb 25 22:57:14 2006
main
main_initbuffer
main_compress
main_initbuffer_L1
main_compress_L1
main_compress_L2
main_compress_getbyte1
main_compress_cl_hash
main_compress_L2_L1
main_compress_L2_getbyte2
main_compress_L2_cl_block
main_compress_cl_hash_L1
main_compress_cl_hash_L2
main_compress_L2_cl_block_cl_hash
main_compress_L2_cl_block_output
main_compress_L2_cl_block_cl_hash_L1
main_compress_L2_cl_block_cl_hash_L2
main_compress_L2_cl_block_output_L1
main_compress_L2_cl_block_output_writebytes1
main_compress_L2_cl_block_output_writebytes2
main_compress_L2_cl_block_output_L1_putbyte
main_compress_L2_cl_block_output_writebytes1_L1
main_compress_L2_cl_block_output_writebytes2_L1

Figure 2. Example of a scope graph hierarchy for a benchmark program (compress).

4. Evaluation of the Mälardalen WCET Benchmarks and Ideas for Future Changes

The Mälardalen WCET benchmarks have been used extensively during their five years of existence. The benchmarks have been used mainly in two ways:

1. For evaluation of WCET algorithms and tools in research papers. The following list gives some examples of papers that have used the Mälardalen WCET benchmarks: [11, 17, 6, 16, 10].
2. For comparisons between WCET tools. A subset[1] of the Mälardalen WCET benchmarks was used during the WCET Challenge 2006 [7, 20] as the standard against which the tools were compared.

The benchmarks have been used as test programs also for other purposes, like dynamic programming [9], migration of real-time tasks [19], and scratchpad memory management [5].

During the years, we have received a lot of feedback. The issues that have been raised mainly belong to some of the categories below. We present the feedback, together with ideas for future changes.

The benchmarks are mostly small programs. The Mälardalen WCET Benchmarks are rather small (all except two are less than 900 LOC). This can be convenient and handy. However, they typically test just a few programming constructs. The small sizes also imply that it can be hard to test how algorithms and tools scale with larger programs. Moreover, they are typically just parts of programs, i.e., they contain a rudimentary main plus some functions. Another drawback is that the whole program often fits in a cache, so it is hard to evaluate cache analyses. Therefore, it would be interesting with larger code sizes constituting full applications.

The benchmarks are not real-time industrial applications. Many of the Mälardalen WCET benchmarks are non-real-time programs, which is acceptable if only different programming constructs need to be tested. But what often is needed is industrial real-time applications with a realistic

[1]The selected programs are marked with an * at the web page.

code size, and a mix of code constructs typical for such applications. However, it seems to be hard to get such applications from the industry, and to get permission to publish the code on an open web site. One possibility is to add benchmarks that was used during the WCET Challenge 2008 (the DEBIE-1 benchmark and *rathijit_1* through *rathijit_4*). These benchmarks are available through the WCET Tool Challenge 2008 homepage [22]. We also would like to get more code examples from industry, and we have an idea how that might be done (see Section 5).

The benchmarks are mainly focussed on flow analysis. What seems to be missing is programs that are targeting testing of program analysis for, e.g., instruction caches, data caches, branch predictions and/or other type of hardware features.

Some program constructs are missing. Even though there are some benchmark programs containing, e.g., unstructured code and recursion, there could be more complex examples to really hard-test such troublesome constructs. Other types of program constructs that could be added is code with highly context-sensitive execution behaviour, programs with complex low-level code (like bit-operations and shifts), use of dynamic memory, mode-specific behaviour, tasks with multiple roots, tasks wrapped in a loop, and programs using function pointers.

Too few benchmarks are multi-path programs. As mentioned above, all current benchmarks are basically single path programs. Therefore, the benchmarks should be extended to include programs with multiple input values. The possible input-value combinations should be an easily available part of the benchmark.

Weak support for measurement-based WCET analysis. The main limitations to using the Mälardalen WCET benchmarks in end-to-end and hybrid measurement-based approaches include the following: the inputs of each program are fixed in the file and therefore different inputs cannot be supplied as parameters; bounds on the input variables are not specified, thus the turnaround time of testing is excessive; the worst-case test vector is not given and thus obtaining the actual WCET is impossible; a common set of realisitic test vectors is missing, thus different tools and techniques are very likely to generate different inputs, making comparison awkward.

Our idea to tackle these problems is to provide measurement-based versions of the benchmarks which consume a test vector from the command line. Furthermore, it is also useful to provide:

- A test harness which calls each benchmark with a predefined (large) set of test vectors. These test data will be generated *a priori* through, for example, a genetic algorithm. The rationale for such a test harness is that it provides a common framework to compare different hybrid measurement-based approaches.
- Bounds on input variables. The key part of end-to-end approaches is the test-vector generation stage, thus merely providing a static set of test vectors is not sufficient. By also supplying bounds on the input variables, therefore, allows an exhaustive exploration of the input space. These bounds are also useful for static analysis tools.

Only C programs are available. It can be considered as a weakness that the current benchmarks only include programs written in C. After all, real-time systems are coded in many other languages, like assembler, C++, Java, and Ada. Also, more code generated from real-time systems modelling tools, like UML and Simulink/MATLAB would be interesting to add to the benchmarks.

Code for parallel systems is missing. Benchmarks containing code for parallel systems where tasks interact might be interesting, as the WCET research moves towards multicore systems.

Precompiled binaries should be available for more types of compilers and processors. At present, there are precompiled binaries available only for the Renesas H8300 processor (in COFF format) using gcc. It would be of interest to have precompiled binaries generated by for the most common processors in real-time systems, including SimpleScalar/M5 configuration files and compiler options. A set of often used compilers should be chosen, and the compilation should be made with a suitable number of basic flag settings.

The benchmark web site should include more results and statistics. It would be interesting for the developers to have more results for the benchmark programs at the web site, for comparison and debugging. Also, available results for, e.g., flow analysis would let researchers concentrate on low-level analysis, and vice versa.

- Actual worst case execution time for different compilers and processors.
- The inputs that provoked the worst case execution time, and the associated path.
- The WCET estimates generated by different tools.
- Flow facts generated by different tools: loop bounds, infeasible paths, recursion depths, etc.
- Results from low-level analysis, like timing for code parts (like basic blocks), cache misses, branch prediction misses, etc.
- Statistics for the programs, like number of functions, function calls, variables, etc.

More types of graphs. There could be more graphs for the benchmarks, e.g., control flow graphs (CFGs), and other graphs generated by the various WCET tools.

5. Conclusion and Future Work

This paper analysed the Mälardalen WCET benchmarks as they exist today. Since their introduction in 2005, they have been used extensively by WCET researchers and developers. The feedback from multiple researchers has highlighted their strengths and existing drawbacks. Taking these onboard, future work will include enhancements to the benchmarks in the following directions: better support for measurement-based analyses; larger programs to stress scalability of tools; more realistic real-time programs and a wider range of languages and code constructs to test applicability of tools. With these upcoming modifications, the WCET benchmark suite will continue to provide a suitable framework for researchers to evaluate their WCET tools and techniques in a multitude of dimensions.

We also propose a new organization of the work with the benchmarks. We need to engage the WCET community of researchers and developers be able to continuously extend the benchmarks to meet the needs of the community. Therefore, we suggest to form a committee with a responsibility for the benchmarks, and that the benchmark web site is transformed to an open wiki, with possibility for the WCET community to easily update the benchmarks and the associated meta-data.

The updates should be supervised by a committee and a steering group. The committee should be responsible for the organization of the benchmarks, the presentation of the benchmarks on the wiki, the quality check of the benchmarks, and the reporting of the state of the benchmarks to the the WCET community.

The committee and steering group should include representatives from different groups, like WCET researchers, tool vendors and real-time systems developers and industry users. The industry representatives could help in getting permission to publish real applications as benchmarks. A broad view of technical and other view should be represented, like measurement, flow analysis and low-level experts, users of small and large systems, hard and soft real-time system developers, etc.

The authors offer hosting this new web site at Mälardalen University.

Acknowledgment

This research was supported by the KK-foundation through grant 2005/0271, and the ALL-TIMES FP7 project, grant agreement no. 215068.

References

[1] ABSINT. aiT tool homepage, 2010. `www.absint.com/ait`.

[2] Homepage for the SPEC CPU2006 benchmark, Apr. 2010. URL: `http://www.spec.org/cpu2006`.

[3] Homepage for the Drystone benchmark, Apr. 2010. URL: `http://www.netlib.org/benchmark/dhry-c`.

[4] Homepage for the EEMBC benchmark, Apr. 2010. URL: `http://www.eembc.org`.

[5] EGGER, B., KIM, S., JANG, C., LEE, J., MIN, S. L., AND SHIN, H. Scratchpad memory management techniques for code in embedded systems without an mmu. *IEEE Transactions on Computers 99*, PrePrints (2009).

[6] ERMEDAHL, A., SANDBERG, C., GUSTAFSSON, J., BYGDE, S., AND LISPER, B. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *Proc. 7th International Workshop on Worst-Case Execution Time Analysis (WCET'2007)* (Pisa, Italy, July 2007), C. Rochange, Ed.

[7] GUSTAFSSON, J. The worst case execution time tool challenge 2006. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[8] Homepage for the Heptane WCET analysis tool, 2006. `www.irisa.fr/aces/work/heptane-demo`.

[9] JAFFAR, J., SANTOSA, A. E., AND VOICU, R. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence* (2008), AAAI Press, pp. 297–303.

[10] KIRNER, R., Ed. *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Prague, Czech Republic, July 1, 2008* (2008), vol. 08003 of *Dagstuhl Seminar Proceedings*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[11] LI, X., LIANG, Y., MITRA, T., AND ROYCHOUDHURY, A. Chronos: A timing analyzer for embedded software. *Science of Computer Programming 69*, 1 - 3 (2007), 56–67.

[12] MÄLARDALEN UNIVERSITY. WCET project homepage, 2010.
`www.mrtc.mdh.se/projects/wcet`.

[13] Homepage for the MiBench embedded benchmark, Apr. 2010.
URL: `http://www.eecs.umich.edu/mibench`.

[14] OTAWA homepage, 2010.
`http://www.otawa.fr/`.

[15] PapaBench homepage, 2010.
`http://www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`.

[16] PRANTL, A., KNOOP, J., SCHORDAN, M., AND TRISKA, M. Constraint solving for high-level WCET analysis. *CoRR abs/0903.2251* (2009).

[17] PUAUT, I., AND PAIS, C. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe* (San Jose, CA, USA, 2007), EDA Consortium, pp. 1484–1489.

[18] RAPITA. RapiTime WCET tool homepage, 2010.
`www.rapitasystems.com`.

[19] SARKAR, A., MUELLER, F., RAMAPRASAD, H., AND MOHAN, S. Push-assisted migration of real-time tasks in multi-core processors. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2009), ACM, pp. 80–89.

[20] TAN, L. The worst case execution time tool challenge 2006: The external test. In *Proc. 2^{nd} International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)* (Paphos, Cyprus, Nov. 2006), T. Margaria, A. Philippou, and B. Steffen, Eds.

[21] TIDORUM. Bound-T tool homepage, 2010. `www.bound-t.com`.

[22] Homepage for WCET Tool Challenge 2008 (WCC'08), 2008.
`http://www.mrtc.mdh.se/projects/WCC08/`.

[23] Mälardalen WCET benchmarks homepage, 2010.
`http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.