TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research*

Heiko Falk[1], Sebastian Altmeyer[2], Peter Hellinckx[3], Björn Lisper[4], Wolfgang Puffitsch[5], Christine Rochange[6], Martin Schoeberl[5], Rasmus Bo Sørensen[5], Peter Wägemann[7], and Simon Wegener[8]

1 **Hamburg University of Technology, Institute of Embedded Systems, Germany**
Heiko.Falk@tuhh.de
2 **University of Amsterdam, The Netherlands**
altmeyer@uva.nl
3 **University of Antwerp, iMinds, Belgium**
peter.hellinckx@uantwerpen.be
4 **Mälardalen University, School of Innovation, Design, and Engineering, Sweden**
bjorn.lisper@mdh.se
5 **Technical University of Denmark, Department of Applied Mathematics and Computer Science, Denmark**
{wopu,masca,rboso}@dtu.dk
6 **University of Toulouse, France**
rochange@irit.fr
7 **Friedrich-Alexander University Erlangen-Nürnberg, Germany**
waegemann@cs.fau.de
8 **AbsInt Angewandte Informatik GmbH, Germany**
wegener@absint.com

Abstract

Engineering related research, such as research on worst-case execution time, uses experimentation to evaluate ideas. For these experiments we need example programs. Furthermore, to make the research experimentation repeatable those programs shall be made publicly available.

We collected open-source programs, adapted them to a common coding style, and provide the collection in open-source. The benchmark collection is called TACLeBench and is available from GitHub in version 2.0 at the publication date of this paper. One of the main features of TACLeBench is that all programs are self-contained without any dependencies on standard libraries or an operating system.

1998 ACM Subject Classification C.3 SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS, Real-time and embedded systems

Keywords and phrases Benchmark, WCET analysis, real-time systems

Digital Object Identifier 10.4230/OASIcs.xxx.yyy.p

1 Introduction

Good, realistic benchmark suites are essential for the evaluation and comparison of worst-case execution time (WCET) analysis, compiler, and computer architecture techniques. TACLeBench provides a freely available and comprehensive benchmark suite for timing

* This work was partially supported by COST Action IC1202 Timing Analysis on Code-Level (TACLe).

Conference/workshop/symposium title on which this volume is based on.
Editors: Martin Schoeberl; pp. 1–11

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

analysis and related research topics. TACLeBench will be continuously extended by novel benchmarks, especially by parallel multi-task/multi-core benchmarks. The extension of TACLeBench will be carefully managed with snapshots and versioning so that it is clear which code has been used in a research experiment. The overall goal is to establish TACLeBench as the standard benchmarking suite for WCET analysis, WCET oriented compiler and computer architecture research worldwide.

TACLeBench is a collection of currently 54 benchmark programs from several different research groups and tool vendors around the world. These benchmarks are provided as ISO C99 source codes. The source codes are 100% self-contained; no dependencies to system-specific header files via `#include` directives or an operating system exist. All input data is part of the C source code. Potentially used functions from math libraries are also provided in the form of C source code. This makes the TACLeBench collection useful for general embedded/barebone systems where no standard library is available.

Furthermore, almost all benchmarks are processor-independent and can be compiled and evaluated for any kind of target processor. The only exception is PapaBench that uses architecture-dependent I/O addresses and currently supports Atmel AVR processors only.

Since TACLeBench addresses the needs imposed by timing analysis tools, all benchmarks are completely annotated with flow facts. These flow facts are directly incorporated into the C source codes using Pragmas. TACLeBench distinguishes between so-called flow restrictions, loop bounds, and entry points. Besides flow restrictions, TACLeBench also uses loop bound flow facts, which simplify the annotation of regular loops. Loop bounds provide an upper and a lower bound for the number of iterations of the annotated loop. Finally, TACLeBench uses entry point annotations that denote points in a program's control flow graph where the control flow may start. Typically, this is the "main" function of a program, but in a (possibly interrupt-driven) multi-task system, there may be multiple entry points in a single set of source files. These entry points may even share some common code. In order to mark such task entries, each function of a multi-task application where a task begins can be marked as an entry point. The complete specification of the used flow fact language can be found in [3], which is part of the source distribution.

If you would like to share your benchmarks with us, feel free to contact Heiko Falk (or any coauthor of this paper) in order to have your source codes included in TACLeBench.

The first version of TACLeBench (version 1.0, available from[1]), which was produced by Heiko Falk, was a collection of 102 programs from several different research groups. We keep this first version tagged with "V1.0" in the public GitHub repository.[2] The version described in this paper is version 2.0 and tagged as such in the repository. The intention is to have the HEAD of the master branch being the most recent, versioned snapshot of TACLeBench. Future development and additions will be performed on a development branch so that HEAD of master is always a consistent snapshot.

This paper is organized in 5 sections: The following section presents related work. Section 3 presents the benchmark collection, its classification, and the updates to make them useful. Section 4 evaluates the benchmark collection. Section 5 concludes.

[1] `http://www.tacle.eu/index.php/activities/taclebench`
[2] `https://github.com/tacle/tacle-bench`

2 Related Work

The Mälardalen WCET benchmarks (MRTC) [6] is the first collection of programs especially intended for benchmarking WCET analysis tools, with a focus on program flow analysis. It was collected from several sources in 2005, and has since then been used in many WCET research projects as well as for the WCET Tool Challenge 2006 [5]. A subset of the Mälardalen benchmarks has even been translated to Java [8]. Most benchmarks are relatively small, except two C programs that have been generated from tools. The benchmarks also contain all input data. This effectively turns them into single-path programs, which makes them less suitable for evaluating tools that can handle multi-path codes. We include most of the benchmarks from the Mälardalen WCET benchmark suite in TACLeBench. We dropped benchmarks where the licensing terms are unknown or even disallow distributing the source.

MiBench [7] is a collection of benchmarks targeting the embedded domain and providing them in open-source. We include some of the MiBench benchmarks, especially those where it was possible to include the input data with the C source.

DEBIE [9] is a program derived from a satellite-mounted detector of micro-meteorids and space debris. It was developed by Space Systems Finland Ltd and was converted by Tidorum Ltd into a portable benchmark for real-time applications. DEBIE is a multi-task application that consists of 8, partially small, different tasks. DEBIE is accompanied by a specification of valid input and output data and of required activation rates of the individual tasks.

PapaBench [10] has been derived from Paparazzi,[3] a project for unmanned aerial vehicles. PapaBench includes two software components that run on separate processors: the *fly-by-wire* part controls the flight while the *autopilot* part controls the GPS and executes the flight plan (which is decided offline). Both software parts cumulate 13 tasks that are subject to precedence constraints and 6 interrupt service routines.

The Embedded Microprocessor Benchmark Consortium (EEMBC) [1] provides a benchmark suite dedicated to the evaluation of the performance of embedded hardware and embedded software. The benchmarks are divided into subset according to the target domain, e.g., the automotive domain, phones and tablets, but also big data and cloud computing. To improve comparability between different systems, the consortium provides a test-harness that allows to derive certifiable scores. The test-harness, being a clear advantage in terms of comparability, constitutes a hindrance in terms of portability and usability. Whereas the TACLeBench has been designed to ease portability and to allow the immediate use of the benchmark with a large variety of tools and platforms, the EEMBC benchmarks are not stand-alone executable without the test-harness. Furthermore, in stark contrast to TACLeBench, the EEMBC benchmarks are not published under an open-source license. Instead, the benchmarks are behind a pay-wall, even for purely academic research.

JemBench [13] is a Java benchmark suite targeting embedded Java platforms. JemBench only assumes the availability of a CLDC API, the minimal configuration defined for the J2ME. The core of the benchmark suite consists of adapted real-world applications. The benchmarks are structured in *micro*, *kernel*, *application*, *parallel*, and *streaming* benchmarks. Micro benchmarks are used to measure short bytecode sequences; kernel benchmarks compute a computational kernel; and application benchmarks are real-world programs restructured as standalone benchmarks. Parallel and streaming benchmarks are intended to explore multicore speedup. Benedikt Huber ported one of the application benchmarks (Lift) to C and we include it in TACLeBench.

[3] `https://wiki.paparazziuav.org/`

Table 1 TACLeBench kernel benchmarks

Name	Description	Code Size (SLOC)	Origin
binarysearch	Binary search of 15 integers	47	SNU-RT
bitcount	Couting number of bits in an integer array	164	Bob Stout & Auke Reitsma
bitonic	Bitonic sorting network	52	MiBench
bsort	Bubblesort program	32	MRTC
complex_updates	Multiply-add with complex vectors	18	DSPStone
countnegative	Counts signes in a matrix	35	MRTC
crc	CRC computation on 40 bytes of data	66	SNU-RT
fac	Factorial function	21	MRTC
fft	1024-point FFT, 13 bits per twiddle	78	DSPStone
filterbank	Filter bank for multirate signals	75	StreamIt
fir2dim	2-dimensional FIR filter convolution	75	DSPStone
iir	Biquad IIR 4 sections filter	27	DSPStone
insertsort	Insertion sort	35	SNU-RT
jfdctint	Discrete-cosine transformation on a 8x8 pixel block	123	SNU-RT
lms	LMS adaptive signal enhancement	51	SNU-RT
ludcmp	LU decomposition	68	SNU-RT
matrix1	Generic matrix multiplication	28	DSPStone
md5	Message digest algorithm	344	NetBench
minver	Floating point matrix inversion	141	SNU-RT
pm	Pattern match kernel	484	HPEC
prime	Prime number test	41	MRTC
quicksort	Quick sort of strings and vectors	992	MiBench
recursion	Artificial recursive code	18	MRTC
sha	NIST secure hash algorithm	382	MiBench
st	Statistics calculations	90	MRTC

3 The Benchmark Collection

3.1 Benchmark Sources and Classification

The benchmarks included in TACLeBench are sourced from single sources and benchmarks collections. The benchmarks are: SNU-RT benchmark suite, MiBench embedded benchmark suite, Mälardalen Real-Time Research Center (MRTC) WCET benchmarks, DSPStone from RWTH Aachen, StreamIt from MIT, NetBench from UCLA, MediaBench, and the HEPC challenge benchmark suite. We have specified the origin of each benchmark in Tables 1-5.

As a measure of the size of each benchmark, we present the number of source lines of code (SLOC). The SLOC count excludes input data arrays and the initialization code. We used the Linux utility `sloccount` to measure the SLOC. The benchmarks are divided into five classes: Kernel, sequential, application, test, and parallel.

The kernel benchmarks, listed in Table 1, are synthetic benchmarks implementing small kernel functions; the size of the kernel benchmarks is in the range of 18 to 992 SLOC. The sequential benchmarks, listed in Table 2, implement large functions block, such as encoders and

Table 2 TACLeBench sequential benchmarks

Name	Description	Code Size (SLOC)	Origin
adpcm_dec	ADPCM decoder	293	SNU-RT
adpcm_enc	ADPCM encoder	316	SNU-RT
ammunition	C compiler arithmetic stress test	2431	Vladimir Makarov
anagram	Word anagram computation	2710	Raymond Chen
audiobeam	Audio beam former	833	StreamIt
cjpeg_transupp	JPEG image transcoding routines	608	MediaBench
cjpeg_wrbmp	JPEG image bitmap writing code	892	Thomas G. Lane
dijkstra	All pairs shortest path	117	MiBench
epic	Efficient pyramid image coder	451	MediaBench
fmref	Software FM radio with equalizer	680	StreamIt
g723_enc	CCITT G.723 encoder	480	SUN Microsystems
gsm_dec	GSM provisional standard decoder	543	MediaBench
gsm_enc	GSM provisional standard encoder	1491	MediaBench
h264_dec	H.264 block decoding functions	460	MediaBench
huff_dec	Huffman decoding with a file source to decompress	183	David Bourgin
huff_enc	Huffman encoding with a file source to compress	325	David Bourgin
mpeg2	MPEG2 motion estimation	1297	MediaBench
ndes	Complex embedded code	260	MRTC
petrinet	Petri net simulation	500	Friedhelm Stappert
rijndael_dec	Rijndael AES decryption	820	MiBench
rijndael_enc	Rijndael AES encryption	734	MiBench
statemate	Statechart simulation of a car window lift control	1038	Friedhelm Stappert
susan	MR image recognition algorithm	1491	MiBench

decoders, that are used in many embedded systems. The size of the sequential benchmarks is in the range of 117 to 2710 SLOC. The sequential benchmarks cover graph search, cryptographic algorithms, compression algorithms, etc. Three artificial test benchmarks, listed in Table 3, are used to stress test WCET analysis tools. The two parallel benchmarks, listed in Table 4, are: Debie and PapaBench. These two benchmarks are comparable in size and in the number of tasks.

The application benchmarks, derived from real applications and provided with simulated input stimuli, are listed in Table 5. Lift is a lift controller that has been deployed at a factory in Turkey. The hardware is based on a Java processor (JOP). The controller has just a few inputs (command buttons and input sensors for the height measurement) and a simple motor control. The I/O devices are simulated in the benchmark. The Java version of Lift is part of the Java benchmark suit JemBench [13]. Benedikt Huber has translated lift to C.

powerwindow implements a controller for an electric window in a car. Both the driver and the passenger are able to control the window by requesting the window to roll up or

Table 3 TACLeBench test benchmarks

Name	Description	Code Size (SLOC)	Origin
cover	Artificial code with lots of different control flow paths	620	MRTC
duff	Duff's device	35	MRTC
test3	Artificial WCET analysis stress test	4235	Universitaet des Saarlandes

Table 4 TACLeBench parallel benchmarks

Name	Description	#Tasks	Code Size (SLOC)	Origin
Debie	DEBIE-1 instrument observing micro-meteoroids and small space debris	8	6615	Tidorum Ltd
PapaBench	UAV autopilot and fly-by-wire software	10	6336	Paparazzi

down. In case an object is stuck between the window and the door frame, the controller will move the window down to avoid damaging the object.

3.2 Issues with the Original Sources

The original benchmarks include all input data or we added input data into the C source. However, this effectively turns them into single-path programs. This fact could be used by analysis tools to explore only this single path. Another consequence of the fixed input data, and that some programs do not provide any return value, is that compilers with optimizations turned on can optimize most of the code away. However, to prohibit the unwanted compiler optimizations we changed the way input data is represented in variables (made them `volatile`), and made the return of `main` dependent on the benchmark calculation.

Some benchmarks contain target dependent code. For example, PapaBench contains hardcoded I/O addresses. Furthermore, a few benchmarks (e.g., rijndael) are byte order dependent and there is no standard way in C to detect the byte order of a processor. Finally, some benchmarks can be executed only once, either because they rely on global initialization, or because they use `malloc` but not `free`.

Table 5 TACLeBench application benchmarks

Name	Description	#Tasks	Code Size (SLOC)	Origin
lift	A lift controller	1	361	Martin Schoeberl
powerwindow	Distributed power window control	4	2533	CoSys-Lab

3.3 Benchmark Updates

The benchmarks have been rewritten to split the functions of input data initialization, the benchmark itself, and computing a return value depending on the output data. Moving the input data generation into its own function resulted sometimes in movements from originally stack allocated data into global data. All function and variable names are prepended with the benchmark name to provide unique names. All loops have been annotated with loop bounds. Moreover, several bugs have been fixed, and compiler warnings have been eliminated. The benchmarks are now ISO C99 compliant. Some benchmarks have been renamed. The original name of a benchmark can be found in the comment header of each benchmark. The library functions used by some benchmarks have been moved to their own files. All source files adhere to a common set of formatting rules that can be found in the git repository (*doc/code_formatting.txt*).

Due to these changes, results obtained with the TACLeBench versions of these benchmarks are not comparable with the original versions of the benchmarks.

3.4 Licenses

An issue we encountered with several benchmarks was that the original source did not include any licensing information. In absence of such information, we had to assume that the copyright holder reserves all rights. Wherever necessary, we contacted the copyright holders to obtain the right to use, modify, and redistribute the benchmarks. In a small number of cases we however discovered that the code was in fact under a license that made the benchmark unusable; the respective benchmarks were consequently dropped from the TACLeBench benchmark suite. All benchmarks in the benchmark suite now contain licensing information, such that future developments do not require tracking down the original authors of the benchmark.

3.5 Usage Recommendations

TACLeBench is released in source form. However, to report results based on TACLeBench, the benchmarks shall not be changed. Furthermore, the version of TACLeBench shall be included in any paper.

If possible, use all benchmarks in your evaluation. One issue with subsetting a benchmark collection is to *selecting the most representative benchmarks* to show an improvement. However, this introduces a bias in the result and is considered scientific misconduct. If you really need to subset a benchmark collection you have two possibilities: (1) use only one class of benchmarks, e.g., the sequential benchmarks or (2) use a random selection, possibly generated by a program.

If some benchmarks have not been used, state the reason for exclusion (e.g., "the tool/target architecture does not support floating point numbers").

3.6 Known Uses of TACLeBench

Although TACLeBench is a relative new collection of embedded benchmarks, we can already list some usage of the collection in research projects. This early adaption of TACLeBench is already a strong indication of the need of such a benchmark collection.

- Compiler optimizations: The origin of TACLeBench is the collection of free programs used for evaluation of the wcc compiler [4].

- Measurement-based analysis: TACLeBench has been used to evaluate the continuous measurement-based WCET estimation approach presented in [2]. The different characteristics of the different benchmarks proved to be very useful to trigger edge cases in the analysis and led to various improvements of the prototype. However, the evaluation for this approach happened before version 2.0 of TACLeBench has been finished, which makes the results non-comparable to the final benchmark collection.
- Hybrid analysis: The hybrid model splits the code of tasks into basic blocks and uses measurements to obtain instruction traces. The challenge of this two-layer hybrid approach is tackling the computational complexity problems within the static analysis and accuracy within the measurements based layer. The TACLeBench is used in the COBRA-HPA (COde Behaviour fRAmework-Hybrid Program Analyser) framework that facilitates evaluation of the different approaches using different block sizes. Furthermore, COBRA-TG (Taskset Generator) uses TACLeBench for schedulability analysis. Different scheduling methodologies can be analyzed in a reproducible way using generated tasksets based on specific application descriptions.
- Hardware design: As the TACLeBench collection is self contained, it leads itself as an easy to use benchmark collection to evaluate computer architecture design in the embedded and real-time domain. We have used version 1.0 for the evaluation of the stack cache design optimized for real-time systems [12].

3.7 Source Access and Compiling the Benchmarks

The benchmarks are hosted on GitHub at `https://github.com/tacle/tacle-bench`. Each benchmark is in its own folder and can simply be compiled with your favorite C compiler with following command:

```
cc/gcc/clang *.c
```

4 Evaluation and Sanity Checks

To evaluate the complexity of the benchmarks and their resilience against compiler optimizations, we executed each benchmark on `pasim`, a cycle-accurate simulator of the Patmos architecture [11]. Each benchmark was compiled using `patmos-clang` with activated compiler optimizations (i.e., -O2). The results of this evaluation are summarized in Figure 1. All execution traces start at the function marked as entry point by a pragma directive in the source code of the benchmark. The execution times range from 305 cycles (binarysearch) up to 1,658,333,567 cycles (test3). To put this into relation, the benchmark program test3 runs approximately for 21 seconds on the Patmos platform assuming a CPU Frequency of 80 Mhz. From this evaluation we make the main observation that TACLeBench consists of both short- and long running benchmarks with a huge variety in execution time. No benchmark in the suite is optimized to a single return statement.

Different members of the TACLe COST action changed the code. Such a collaborative procedure is inherently error-prone. Human errors can occur and the original sources were often faulty to start with. The distributive work on the benchmarks led to an additional source of error detection: a benchmark behaving well under system configuration A may be faulty under configuration B. To ensure the quality of the benchmarks and to improve portability, we have thus implemented automatic sanity checks.

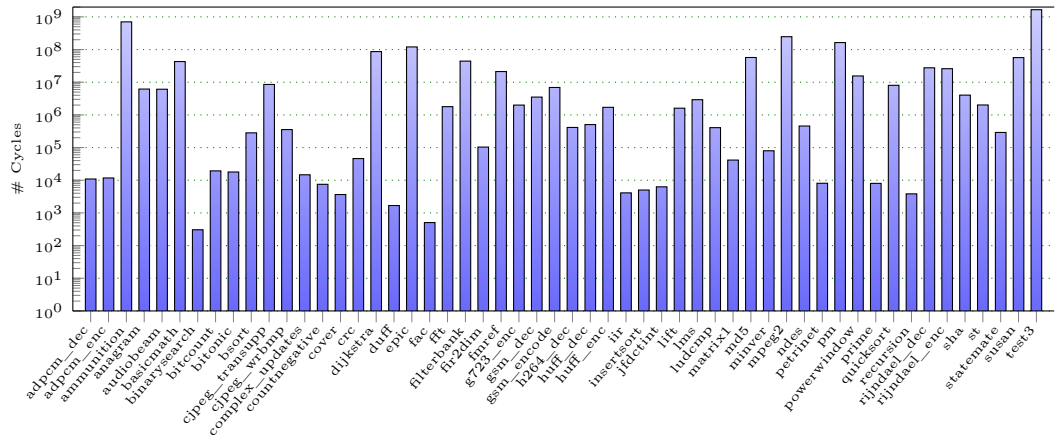

Figure 1 Execution times of programs in TACLeBench range from 305 cycles up to more than 1,600,000,000 cycles on the Patmos architecture.

The code quality is validated centrally and the latest results can be viewed online.[4] For this sanity check, all benchmarks are compiled using *gcc*, *g++* and *clang*, executed and the return values are checked against the expected value. Clang's static analyzer[5] is used to further reveal programming bugs, such as out-of-bounds errors, and also to validate the compliance of the code with the code-formatting rules.

The portability has been checked via a shell-script (`checkBenchmark.sh`) that is now part of the TACLeBench repository. The script allows to quickly identifying incompatibilities of the benchmarks with specific operating systems, compilers, and system configurations. Even though full coverage of all system configurations can never be achieved, we were able to cover most of the common operating systems and compilers.

5 Conclusion

Research in the field of embedded real-time systems needs benchmarks to evaluate research ideas. TACLeBench provides an open source collection of 54 benchmarks. As all benchmarks are self-contained, they are easy to use in systems that are lacking the standard library or an operating systems. This paper reports on the TACLeBench version 2.0. We intend to grow the collection of programs and coordinate the release of benchmark versions. We further welcome contributions to the benchmark collection.

Acknowledgements We want to thank Bendikt Huber for porting the Lift benchmark from Java to C. We want to thank Niklas Holsti from Tidorum Ltd for contributing DEBIE in open-source. We want to thank Yorick De Bock, Florian Kluge, Jörg Mische, and Haoxuan Li for helping in restructuring the benchmarks.

References

1 Edn embedded microprocessor benchmark consortium. `http://www.eembc.org.`

[4] `https://www4.cs.fau.de/Research/TACLeBench`
[5] `http://clang-analyzer.llvm.org/`

2 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume ?? of *OpenAccess Series in Informatics (OASIcs)*, pages ??–??, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

3 Heiko Falk, Timon Kelter, Robert Pyka, and Daniel Schulte. TACLeBench Flow Facts Documentation. White paper, September 2013.

4 Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a wcet-aware c compiler. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

5 Jan Gustafsson. The worst case execution time tool challenge 2006. In Tiziana Margaria, Anna Philippou, and Bernhard Steffen, editors, *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods (ISOLA'06)*, pages 233–240, Paphos, Cyprus, November 2006. IEEE.

6 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

7 Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th annual Workshop on Workload Characterization*, 2001.

8 Trevor Harmon, Martin Schoeberl, Raimund Kirner, and Raymond Klefstad. A modular worst-case execution time analysis tool for Java processors. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 47–57, St. Louis, MO, United States, April 2008. IEEE Computer Society.

9 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the debie software. *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP*, 457:307–312, 2000.

10 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

11 Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

12 Martin Schoeberl and Carsten Nielsen. A stack cache for real-time systems. In *Proceedings of the 18th IEEE Symposium on Real-time Distributed Computing (ISORC 2016)*, York, United Kingdom, May 2016. IEEE.

13 Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*, pages 120–127, New York, NY, USA, August 2010. ACM.