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

Heiko Falk¹, Sebastian Altmeyer², Peter Hellinckx³, Björn Lisper⁴, Wolfgang Puffitsch⁵, Christine Rochange⁶, Martin Schoeberl⁷, Rasmus Bo Sørensen⁸, Peter Wägemann⁹, and Simon Wegener¹⁰

¹ Hamburg University of Technology, Institute of Embedded Systems, Hamburg, Germany  
Heiko.Falk@tuhh.de  
² University of Amsterdam, Amsterdam, The Netherlands  
altmeyer@uva.nl  
³ University of Antwerp, iMinds, Antwerp, Belgium  
peter.hellinckx@uantwerpen.be  
⁴ Mälardalen University, School of Innovation, Design, and Engineering, Västerås, Sweden  
bjorn.lisper@mdh.se  
⁵ Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark  
wopu@dtu.dk  
⁶ University of Toulouse, Toulouse, France  
rochange@irit.fr  
⁷ Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark  
masca@dtu.dk  
⁸ Technical University of Denmark, Department of Applied Mathematics and Computer Science, Lyngby, Denmark  
rbos@dtu.dk  
⁹ Friedrich-Alexander University Erlangen-Nürnberg, Erlangen-Nürnberg, Germany  
wagemann@cs.fau.de  
¹⁰ AbsInt Angewandte Informatik GmbH, Saarbrücken, 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 1.9 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.WCET.2016.2

* This work was partially supported by COST Action IC1202 Timing Analysis on Code-Level (TACLe).
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 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 53 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 from1), 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 1.9 and tagged as such in the repository. Version 2.0 will be soon available when the last missing programs have been formatted. 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.

---

1 http://www.tacle.eu/index.php/activities/taclebench
2 https://github.com/tacle/tacle-bench
Section 3 presents the benchmark collection, its classification, and the updates to make them useful. Section 4 evaluates the benchmark collection. Section 5 concludes.

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-meteoroids 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, 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) [2] provides a benchmark suite dedicated to the evaluation of the performance of embedded hardware and embedded software. The benchmarks are divided into subsets 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 deriving 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
Table 1 TACLeBench kernel benchmarks.

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

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 The Benchmark Collection

3.1 Benchmark Sources and Classification

The benchmarks included in TACLeBench are sourced from single sources and benchmark 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.
Table 2 TACLeBench sequential benchmarks.

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

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 function blocks, such as encoders and decoders, which 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
Table 3 TACLeBench test benchmarks.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Code Size (SLOC)</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>cover</td>
<td>Artificial code with lots of different control flow paths</td>
<td>620</td>
<td>MRTC</td>
</tr>
<tr>
<td>duff</td>
<td>Duff’s device</td>
<td>35</td>
<td>MRTC</td>
</tr>
<tr>
<td>test3</td>
<td>Artificial WCET analysis stress test</td>
<td>4235</td>
<td>Universität des Saarlandes</td>
</tr>
</tbody>
</table>

Table 4 TACLeBench parallel benchmarks.

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

class. 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 down. In case an object is stuck between the window and the doorframe, 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.

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 bench-
mark 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 [1]. 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:

```
c/c/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. 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 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-formating 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 53 benchmarks. As all benchmarks are self-contained, they are easy to use in systems that are lacking the standard library or an operating systems. 2.0 will be released when the last missing programs have been adapted to the common coding style. 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  Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In

4  https://www4.cs.fau.de/Research/TACLeBench
5  http://clang-analyzer.llvm.org/
TACLeBench: A Benchmark Collection to Support WCET Research


