A Benchmarking Suite for Measurement-Based WCET Analysis Tools*

Sven Bünte
Technische Universität Wien
1010 Wien, Austria
sven@vmars.tuwien.ac.at

Michael Tautschnig
Technische Universität Darmstadt
64289 Darmstadt, Germany
tautschn@model.in.tum.de

Abstract

Worst case execution time analysis based on measurements requires large test suites to obtain reliable numbers. We are thus developing tools to efficiently generate these test sets in a whitebox-testing approach.

To make project progress measurable and guard against regressions, a benchmarking suite is sought for. We present a set of requirements that have been collected and outline the design of a benchmarking tool-set. While we are currently developing our domain-specific tool chain, we assume the presented architecture to be sufficiently general.

1 Introduction

In the FORTAS project [1] we are concerned with execution time analysis of embedded software, focusing in particular on control software written in C. We try to satisfy the industrial demand for a method and a tool that (a) is more reliable and predicable than ad hoc testing, (b) requires little knowledge of the target hardware, and (c) is fully automated.

Unlike the static analysis based approach [3, 4], our measurement based [6] analysis requires large test suites to obtain reliable results. We are thus developing tools to efficiently generate these test sets in a whitebox-testing approach.

In developing such a tool we require two kinds of benchmark suites. First, effective testing of the resulting tool must be supported, guarding against regressions and performance degradations. Second, a data base of C programs has to be acquired to quantify and compare our measurement results. In this paper we describe a set of key requirements that have been identified during the design of our benchmark suite and its related tools.

*Supported by DFG/FWF grant FORTAS – Formal Timing Analysis Suite for Real Time Programs (VE 455/1-1).

2 Requirements Identified in FORTAS

The FORTAS project focuses on analyzing embedded control software written in C (including machine code fragments). We are especially interested in structural aspects of the program under test, e.g., loop structures or the amount and nesting depth of conditional statements. Furthermore, several target platforms and compilers should be supported. User information about the program under test, such as loop bounds, have to be incorporated into the analysis.

The performance analysis of the framework in terms of time and memory usage, as well as the accuracy of computed resource consumptions (time and power) turn out to be the main requirements for our benchmark suite. In order to perform functional validation as well, regression tests must be provided, too. This calls for storing expected output behavior and to relate it accordingly.

This persistence should also provide means to compare results of different tools of the FORTAS tool chain and/or different versions thereof. These results comprise the estimated worst-case execution time, or the gained code-coverage with respect to different metrics (statement coverage, condition-coverage, MC/DC, ...). Another part of the result is generated test cases for the benchmark. Those might be useful for a subsequent analysis of another program under test (e.g., a slightly different version of the benchmark). Therefore, these test cases must be referenced. The presented requirements so far entail a need for versioning of benchmarks on the one hand and their results on the other hand.

The user should be able to select specific subsets of all benchmarks in the suite. To allow for an effective implementation of the search, tags may be attached to each benchmark. These tags may be user-defined (e.g., application domain of benchmark, intended compilers or target platforms, author name, ...), but also results of automated analysis of the code if conceivable (programming language, loop structures, lines of code). The reflected properties may also be custom tags. Another feature could be a guided semi-automatic tagging procedure, where the user is asked about

properties like "Where did you get the benchmark from?" with predefined options to select from.

Due to the distributed nature of the project, also the test suite must support collaborative and distributed work. Security is a main issue here, since the non-private links will be involved. Some benchmarks might be subject to non-disclosure agreements for which user privileges must be administrated.

3 Sharing Benchmark Suites

When planning the benchmarking suite, we observed that locally various research groups had already collected or created benchmark sets to derive performance properties of their individual tools and approaches. These sets turned out to be applicable for other research groups as well. Consequently, the design of a benchmark suite that reasonably abstracts from individual needs will enhance productivity. We thus define our first goal: The suite must be *sufficiently generic*.

Yet, research of prior art on published benchmark suites yielded only few results [2, 5, 7, 9]. We thus gather that a way for *publishing and exchanging* benchmarks must be found. While intellectual property may be an issue in industrial corporations, our local research led us to the assumption that a large number of benchmark sets do exist. These could be made available, but only very few of them are published. Still, to support industrial collaborations involving non-disclosure agreements, a benchmark provisioning system must offer both *authentication* and *authorization*. Based on this identity information, different levels of access and privacy may be configured. Of course, this also requires secure remote access.

Towards a Possible Architecture

We aim at a set of tools and the necessary environment to allow effective benchmarking of the FORTAS tool chain. The properties to be checked of the system under test are (1) *completeness of features* as described in our internal architecture document and (2) *absence of regressions* from earlier versions of the tool chain. Further, (3) *performance tests* in terms of time consumption and memory usage, both in absolute values and relative to that of our competitors.

To enable this, benchmark results must be *stored*. That is, both *tool output* and *measurable results*, depending on domain specific metrics, must be retained. First, such a persistence layer must provide *versioning* to retain the *history* of benchmarking. Second, a proper frontend with support for *queries over the results* must be provided. Besides native tools, a high-level interface would be nice in the face of usability. Third, we expect a high number of benchmarks, which incurs large result sets. Thus *scalability* is of ultimate concern. As such, current relational database systems are—pending benchmarking—presumably appropriate. Thereby we also satisfy additional requirements: A user may add a set of *domain specific tags* to benchmarks for later selection of subsets. Further, *overviews and summaries* shall be provided, given certain selection criteria. We stress that both requirements are very involved with the domain and must be highly configurable by the user. Even more so, two users of the suite may be interested in different aspects of the benchmark, thus the tagging shall support user-specific tags as well.

4 General Applicability

We figured out that requirements for a benchmark suite can be generalized for our local research groups. Concerning the matter of public convenience, a generic design is therefore essential. We claim that the ability to assign tags to benchmark programs and results meets this major requirement. Tools can assure that these tags are set according to common standards. However, the user is still free to create and set tags individually.

Another advantage of this approach is that once domain-specific constraints on tags are defined and ordered hierarchically, tools that verify the compliance of these standards can be applied in the very same order. As an example, at first the user is guided by a tool to set very general attributes like *intended application domain* or *benchmark source*. Then another tools takes care that more specific properties are specified, e.g., *programming language*. This procedure is to be continued until all standards have been considered.

5 Proposed Architecture

To achieve the goals outlined above, we build our framework upon two distinct databases, as shown in Fig. 1. First, benchmark suites and test results are kept in a subversion repository. This (a) caters for versioning of both benchmarks and results, (b) is well suited for the expected major part of plain text data, (c) supports copying and modifying benchmarks while retaining their history with little overhead, and (d) provides *hooks* for automated updates of an associated relational database. This database is used to keep track of tags and benchmark summary information, which can then be queried efficiently. Upon a query, if necessary, a set of benchmark identifiers is returned, such that the corresponding benchmarks can then be pulled from the subversion repository. It is conceivable that tags are kept in the subversion repository as well, which would allow for a database rebuild at any time.

We note that both subversion and current relational database implementations provide authentication and au-

thorization mechanisms, including different levels of access. Further, secure transport using HTTP over SSL is easily implemented. Following recent trends we might even consider the benchmarking suite as a web service.

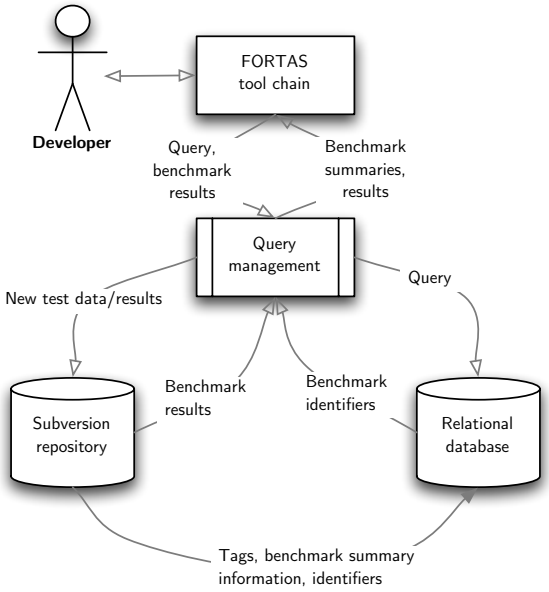

Figure 1. Sketch of the proposed architecture

The workflow, as depicted in Fig. 1, is envisioned as follows. Once a test run has finished, the results and the involved benchmarks are committed to the subversion repository in an automated fashion. This commit includes environmental information, such as the version of the tool chain under scrutiny, and possible user-defined tags. Further, in regression tests using the diagnostics framework [8], conformance with expected results is tested for and stored in the repository. Apart from this it is still to be determined which properties of test suites and test results require user interaction to define the values stored in the database.

6 Conclusions

We presented the current status of the benchmarking tool chain to be used in the FORTAS project. Even though some of our requirements may be specific to the domain of measurement-based timing analysis, we presented an architecture that we consider sufficiently generic to be deployed in many kinds of software testing toolkits.

Acknowledgments. We are grateful to Andreas Holzer, Raimund Kirner, Bernhard Rieder, Christian Schallhart, Helmut Veith, Ingomar Wenzel, and Michael Zolda for discussions on the topic of this paper.

References

[1] The FORTAS project, a Formal Timing Analysis Suite. `http://www.fortastic.net`, 2007.

[2] Embedded microprocessor benchmark consortium. `http://www.eembc.org/`.

[3] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems — The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, 1999.

[4] C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[5] IRIT. Paparazzi: Unmanned Aerial Vehicle Software. `www.irit.fr/recherches/ARCHI/MARCH/rubrique.php3?id_rubrique=97`.

[6] R. Kirner, P. Puschner, and I. Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. In *Proc. 4th Euromicro International Workshop on WCET Analysis*, pages 67–70, June 2004.

[7] Mälardalen Research and Technology Centre. The Worst-Case Execution Time (WCET) analysis project. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[8] C. Schallhart. Diagnostics – A framework for logging, unittesting, and runtime diagnostics. `http://packages.debian.org/diagnostics`, 2008.

[9] University of Michigan. MiBench Version 1.0. `http://www.eecs.umich.edu/mibench/`.