COMPILER SUPPORT FOR MEASUREMENT-BASED TIMING ANALYSIS [1]

Raimund Kirner[2] and Michael Zolda[3]

Abstract
Measurement-based timing analysis (MBTA) techniques have been developed as a complimentary to static WCET analysis, in order to exploit worst-case execution time (WCET) analysis at early stages of system development. The direct advantage of MBTA is that, in contrast to static WCET analysis, no timing model of the hardware platform has to be developed. Instead the timing model is generated automatically by performing systematic execution time measurements. MBTA provides high retargetability, as the test suite used for execution time measurements is typically derived from the source code of the program. In order to provide an accurate WCET estimate, the test suite has to provide a sufficient coverage of the temporal system behavior. Here also the compilation tool chain is important as the compiler may introduce additional control flow that is not visible at the source code. In this paper we present FORTAS, an MBTA tool that systematically generates test data using a range of different techniques, like heuristics and model checking. Furthermore, we show how compiler-support for MBTA can provide code optimization while preserving the code coverage achieved by the MBTA test suite at source-code level. First evaluations indicate that the performance penalty for ensuring coverage preservation of the test suite is low.

1. Introduction

The correct operation of real-time system demands to ensure that the worst-case timing costs of individual actions are compliant with the timing constraints imposed by the environment and by the hardware platform. To ensure this, we have to determine the worst-case execution time (WCET) of all the relevant activities.

The WCET can be determined by static WCET analysis, which is based on static program analysis of the machine code requiring a correct cost model for the hardware platform. Another approach that is becoming increasingly popular in the research community is measurement-base timing analysis (MBTA), where the cost model used to search for a WCET estimate is calibrated by systematic execution-time measurements [12, 13, 2]. Both approaches have their specific merits. With static WCET analysis one can construct a proof for an upper bound of the WCET, based on the assumption that the constructed cost model is fully correct. With MBTA once can avoid the high retargeting cost of the analysis to a new hardware platform, but it is in general not possible to provide a proof that the WCET estimate is safe, i.e., is an upper bound of the WCET. Furthermore, static WCET analysis can result in quite high overestimations and do require a significant effort on code annotations, even at machine-code level.

[2]University of Hertfordshire, Department of Computer Science, United Kingdom, r.kirner@complang.tuwien.ac.at
[3]Vienna University of Technology, Institute of Computer Engineering, Austria, michaelz@vmars.tuwien.ac.at

[1]This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research projects "Sustaining Entire Code-Coverage on Code Optimization" (SECCO) under contract No P20944-N13 and "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13, and by the European Union within the IST FP-7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" under contract no IST-2010-248828.

The preciseness of the MBTA approach heavily depends on the coverage achieved by the test suite at the execution time measurements. One of our research subject is to find heuristics for generation of test data that achieve a good coverage of the systems timing behavior [13].

In this paper we focus on a challenge for MBTA, the traceability of structural code coverage achieved by a test suite at source code down to machine code during optimizing compilation [6]. In Section 2 we describe the principle of our MBTA tool FORTAS. In Section 3 we summarize the calculation of so-called structured code-coverage profiles (SCCP), for which their integration into a compiler is described in Section 4. Section 5 provides a performance evaluation of the SCCP technique and Section 6 concludes the paper.

2. The Formal Timing Analysis Suite

The *FORmal Timing Analysis Suite (FORTAS)* is the research vehicle for our work on MBTA. Designed as an extensible, modular, distributed, concurrent framework, the most important use-case of FORTAS is to obtain early WCET estimates. Given an existing piece of code, an engineer can employ the analysis tools to obtain an estimate of the code's WCET on the target platform of his choice. Doing so can help him in making design decisions concerning the software and/or the intended target platform.

The FORTAS approach is not intended to always provide a safe WCET estimate—a guaranteed upper bounds of the WCET—as is the case in static WCET analysis. Rather, our tools have intentionally been designed to be used in situations where rough (and possible unsafe) estimates are favored considering the major downsides of static WCET analysis: high initial costs to develop a dependable analysis for a specific target platform, no or limited retargetability, and highly pessimistic WCET bounds in the case of modern target platforms.

By design, our approach does not make any limiting assumptions about the target platform: The analysis tools execute the code under scrutiny on the target platform to obtain time-stamped execution traces. The information contained in these traces is then combined with control flow information obtained via traditional code analysis, and a timing model is created. This timing model is then specific to both, the code under scrutiny and the target platform.

The FORTAS approach is highly retargetable: The only component that needs to be provided to support a new target platform is a measurement backend that executes the code under scrutiny with given input data and returns the corresponding time-stamped execution trace. The analysis components can then make use of the new backend to probe the temporal behavior of the code under scrutiny on the new target platform by generating appropriate suites of input data and integrating the learned information into the timing model.

A novel feature of FORTAS is adaptive, iterative refinement of the timing model: Using a feedback mechanism that periodically inspects the current timing model and generates new coverage goals for subsequent input data generation, the tools can adaptively refine the timing model, such that more and more precise WCET estimates can be obtained. During this iterative process, the engineer can, at any time, obtain a WCET estimate from the timing model in its most recent state.

Figure 1 summarizes the workflow of an adaptive WCET analysis with FORTAS. In the following we briefly explain the individual steps in this workflow. Internals of the used methods can be found in [14, 4].

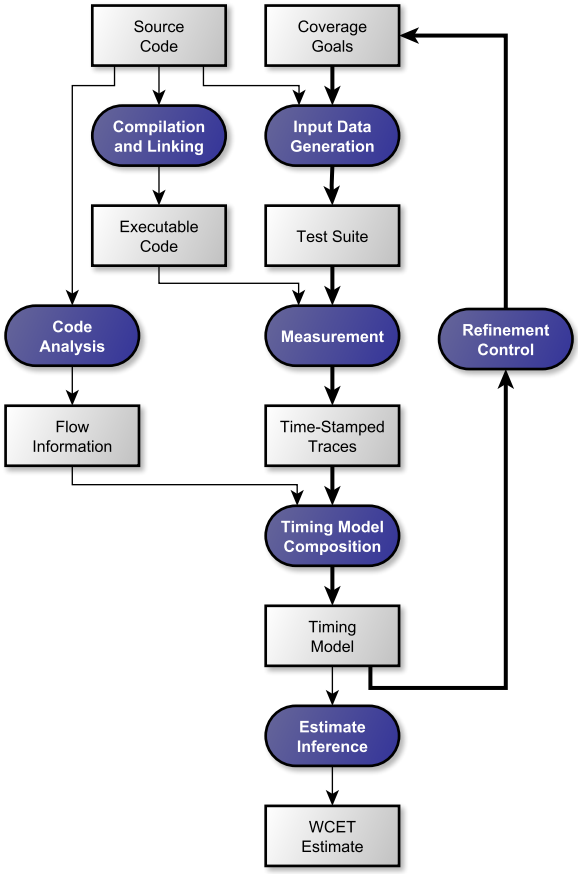

Figure 1. Basic Workflow of FORTAS

Code Analysis performs static code analysis on the code under scrutiny to obtain flow information, most importantly the control flow graph (CFG). Code analysis can also derive loop iteration constraints. In their current development stage, the FORTAS tools require manual specification of loop iteration constraints.

Compilation and Linking takes the code under scrutiny and produces executable code for the target platform. Currently the FORTAS tools are able to process C source code, but the approach could also be used to analyze intermediate or machine code.

Input Data Generation is a central part of MBTA in general, and FORTAS in particular. This step produces a set of suitable input vectors—a *test suite*—to be used in the subsequent measurement step. *Suitable* here means that the test suite should conform to a given *coverage goal*. FORTAS currently supports two different kinds of coverage goals: The first kind are structural coverage goals, e.g., requirements like basic block, condition or decision coverage, or more sophisticated specifications [5, 3]. The second kind are optimization goals, e.g., maximization of locally observed execution times [4]. Concerning the applied techniques for input data generation, the FORTAS tools currently rely primarily on model checking and genetic algorithms.

Measurement performs at least one run of the executable code on the target platform for each input vector from the test suite and produces corresponding *time-stamped execution traces*. A time-stamped execution trace indicates the exact execution sequence of the individual CFG nodes and the execution duration for each entry in this sequence. The current FORTAS implementation supports non-intrusive capturing of end-to-end time-stamped execution traces on the TriCore 1796 processor via a *Lauterbach PowerTrace* [1] device.

Timing Composition combines information from the obtained time-stamped execution traces with the flow information obtained from code analysis into a *timing model* that summarizes the temporal behavior of the code under scrutiny on the target platform.

Refinement Control examines the timing model and generates new coverage goals to be used during subsequent input data generation. The objective of refinement is to increase the precision of the timing model.

Estimate Inference produces a WCET estimate from the timing model. FORTAS currently provides *context-sensitive IPET* [14] as core inference technique.

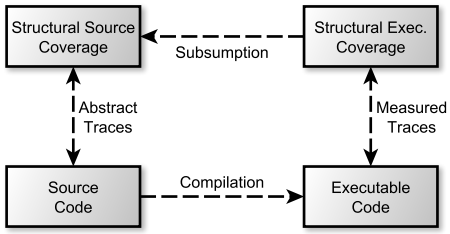

Figure 2. The necessary Relationship between Structural Coverage on two different Code Levels to ensure Coverage Preservation

We have designed FORTAS to generate test suites at the source code level. The advantage of this design is that test data generation is independent of the target platform. We need only one implementation of each generation method to obtain support for all FORTAS platforms.

Our test data generators currently process C source code. In particular, we are using the model-checking based test-suite generator FSHELL [5], to create test suites from structural test goals (see above).

On the other hand, measurement must always be performed at the executable level. It is therefore important that the compilation step preserves the intended structural coverage of the test suite, i.e., the structural coverage achieved by the associated set of abstract traces on the source code level must be subsumed by the corresponding set of measured traces on the executable level. Figure 2 illustrates the necessary relationship between coverage on the two different code levels.

3. Compiler Support for MBTA

To prove that a structural coverage criterion is preserved during optimizing compilation, we have done several steps. First, we have formalized the meaning of the structural code coverage criteria we want to preserve and based on that we have derived corresponding formal coverage preservation criteria. Furthermore, we formalized the code transformations at a suitable abstraction level. Based on that we infer the SCCP profiles, which in essence is a table that indicates for each code transformation whether it does ensure preservation of the structural code coverage criteria.

Table 1 shows an example of SCCP profiles for statement coverage (SC), condition coverage (CC), and decision coverage (DC). As code transformations we have chosen condition reordering (where it is interesting to note that there it makes a difference whether the language provides short-circuit evaluation for the conditions), loop peeling (splitting the iteration space of a loop into multiple parts), and loop inversion (converting a while loop into a do/while loop). For example, the table shows that

Code Optimization	Coverage Preservation		
	SC	CC	DC
Condition reordering (without short-circuit)	✓	·	✓
Condition reordering (with short-circuit)	✓	✓	✓
Loop peeling	·	·	·
Loop inversion	✓	·	·

Table 1. Calculated Structural Code Coverage Preservation Profiles

loop inversion still preserves SC, but not CC or DC. More details of how such SCCP profiles are calculated can be found in [7]. The derivation of preservation criteria for structural code coverage metrics is described in [6].

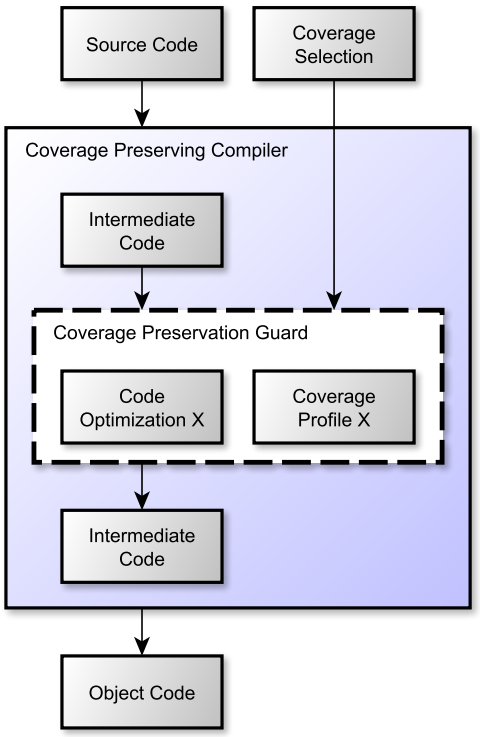

Figure 3. Application of an SCCP Profile

The concept of integrating SCCP profiles into a compiler to preserve selected code coverage criteria is shown in Figure 3. The SCCP profiles have to be realized as guards that enable only those code optimizations for which the chosen coverage criteria are preserved. More details on the concrete implementation are given in the following section.

4. Implementation of the SCCP-Compiler

Once the *structural code coverage profiles* (SCCP) profiles have been calculated, it is relatively easy to integrate them into a compiler. In the following we describe the integration of the SCCP approach into the GCC version 4.5.2 [10, 9]. GCC has been chosen because of its open source license and also for its support of numerous target platforms, including some used in embedded computing.

The SCCP profiles can be thought as a table that indicates by a flag for each code optimization and each code coverage metrics, whether the metrics is preserved by performing the code optimization,

like the example given in Table 1. The real implementation, however, needed some more effort, as GCC supports quite a lot of code optimizations. GCC provides some generic optimization flags `-O0 ... -O3`, where `-O0` means virtually no code optimizations are performed, and `-O3` means the activation of many code optimizations. Still, `-O3` does not mean that all code optimizations in GCC are to be activated, because some code optimizations are not generally beneficial, as, for example, they can significantly increase code size or produce code that improves performance for a limited set of code structures. For example, an optimization that needs explicit activation is *loop unrolling*. GCC also allows to to individually enable/disable each of the code optimizations controlled by the options `-O0 ... -O3`. We have decided to implement support of SCCP profiles for all those code optimizations effected by `-O0 ... -O3`, and also loop unrolling, which resulted in total in 62 code transformations.

The user interface of the compiler has been changed by two extra command-line options in order to control the SCCP features:

–sccp-enforce = metrics-list is the primary option to tell the compiler that structural code coverage preservation has to be considered for the comma-separated list of structural code coverage metrics. Currently, this list can include the items `SC` (statement coverage), `cc` (condition coverage), `dc` (decision coverage), `mcdc` (modified condition-decision coverage), and `pc` (path coverage). For example, to tell the compiler to focus on the preservation of CC and DC, it has to be called with the following option: `--sccp-enforce=cc,dc`.

–sccp-warn-mode is the secondary command-line option, which, in case that the SCCP option `--sccp-enforce=`⟨`metrics-list`⟩ is given as well, tells the compiler to be switched to a passive coverage preservation mode, where no code optimizations are guarded, but instead warnings are emitted, which inform the user that the specified coverage metrics may have been disrupted. If this option is omitted, than the default mode of structural code coverage preservation is to disable in the compiler all those code optimizations that have the potential to disrupt the coverage metrics specified in ⟨`metrics-list`⟩.

At the current stage of implementation the functionality behind the option `--sccp-warn` has not been fully implemented, as experiments so far haven't shown real need for it. In fact, the option `--sccp-warn` would have been useful, if the enforcement of the SCCP mode by disabling code optimizations had a significant reduction of performance of the generated code. The command line interface of GCC has been extended by adding the additional options into the file *common.opt*, from where an existing script of GCC automatically generates the code for parsing and basic handling of the options [9].

```
struct sccp_coverage_profile_type
{
  enum sccp_code_trans trans;    /* id of code optimization */
  unsigned int preserves_mask;   /* each bit set represents a preserved
                                    metrics */
  int *pflag;                    /* address of enabling flag of
                                    optimization */
  const char *trname;            /* name of code optimization */
};

extern struct sccp_coverage_profile_type sccp_coverage_profile[];
```

Figure 4. Added Source Code in GCC to define SCCP Profiles

Having defined the user interface for the SCCP approach, we now demonstrate that it is rather easy to integrate the SCCP behavior in the compiler. Figure 4 shows the definition of the data structure to hold the SCCP profiles. `trans` is an integer value that specifies the numerical identifier of each code transformation. This list of identifiers for all supported code optimizations has been added as an enumeration type in ISO C. The variable `preserves_mask` is a bitmask where a one at a bit position indicates that the corresponding coverage metric is preserved by the code transformation. `pflag` is a pointer to the control variable of the optimization; it is used to disable code optimizations in case that they do not preserve the requested coverage metrics. The entry `trname` holds the name of the optimization, which is used for optimization logs.

```
/* Definitions for structural code coverage preservation (SCCP) */
struct sccp_coverage_profile_type sccp_coverage_profile[] = {
    ...
    { F_INLINE_SMALL_FUNCTIONS, /* min -O2 */
      SCCP_NONE,
      &flag_inline_small_functions, "-finline-small-functions"},
    { F_GLOBAL_CSE, /* min -O2 */
      SCCP_SC | SCCP_CC | SCCP_DC | SCCP_MCDC | SCCP_PC,
      &flag_gcse, "-fgcse"},
    { F_MINOR_EXPENSIVE_OPTS, /* min -O2 */
      SCCP_SC | SCCP_CC | SCCP_DC | SCCP_MCDC | SCCP_PC,
      &flag_expensive_optimizations, "-fexpensive-optimizations"},
    { F_DO_CSE_AFTER_LOOP_OPTS, /* min -O2 */
      SCCP_SC | SCCP_CC | SCCP_DC | SCCP_MCDC | SCCP_PC,
      &flag_rerun_cse_after_loop, "-frerun-cse-after-loop"},
    { F_SAVE_REGS_AROUND_CALL, /* min -O2 */
      SCCP_SC | SCCP_CC | SCCP_DC | SCCP_MCDC | SCCP_PC,
      &flag_caller_saves, "-fcaller-saves"},
    ...
};
```

Figure 5. Excerpt of Source Code added in GCC to initialize SCCP Profiles

The code for the initialization of the SCCP profiles is shown in Figure 5, which shows the initialization for five out of the 62 code optimizations supported by our implementation in GCC. For example, the first entry describes the entry for a special form of function inlining, i.e., the body of subroutines of up to a certain size will be copied into the place of where the function call has been given. Here `F_INLINE_SMALL_FUNCTIONS` is the numeric identifier of the code optimization. `SCCP_NONE` is the bitfield which defines for each structural code coverage metrics whether it is preserved by that code optimization. In the case of macro `SCCP_NONE` is says that none of the coverage metrics is be guaranteed to be preserved. `&flag_inline_small_functions` is the name of the code flag used by GCC to control whether this optimization is enabled or not. Thus, when evaluating the SCCP arguments, this flag will be modified if the corresponding code optimization needs to be disabled to ensure coverage preservation. The entry `-finline-small-functions` represents the corresponding name of the command-line option used to enable/disable this optimization in GCC. The comment `/* min -O2 */` simply means that this code optimization is by default enabled in a general code optimization level of `-O2` or higher has been selected.

The code examples above show the basic mechanism of how we implemented the SCCP profiles in GCC. For other compilers we expect that the integration is of similar ease. However, what made the implementation in GCC nice to test is the ability to control the activation of any code optimization individually. The source code of the prototype implementation of the SCCP compiler can be downloaded from the homepage of the SECCO project [11].

5. Evaluation

The main question behind the SCCP method is how much performance does it cost in order to guarantee preservation of structural code coverage. In this section we describe some first experiments with different settings of optimization levels in GCC 4.5.2. The benchmarks we used are taken from the standard worst-case execution time (WCET) benchmark suite assembled by the Mälardalen University [8]. The Mälardalen WCET benchmarks have been slightly modified in order to show the effects of different optimization levels. This modification consists of adding a new entry routine that calls the original entry routine several times. With this modification the program execution times become a more dominant timing contribution compared to the program call overhead. The experiments were run on a Mac OS-X 10.6 machine with an Intel Core 2 Duo processor running at 3.06 MHz. The execution time measurements where done for different optimization settings of the compiler, using a measurement accuracy of 1ms. The different optimization settings are listed in Table 2. The first four settings are the different generic optimization levels of GCC. In setting FULL we added loop unrolling as an optimization, which is not activated automatically by `-O3`. The setting SCCP1 also uses `-O3` as optimization level, but activates the SCCP enforcement of statement coverage. In setting SCCP2 we enforce the preservation of all currently supported code coverage metrics: statement coverage, condition coverage, decision coverage, MCDC coverage, and path coverage. For both SCCP settings we have chosen `-O3` as the optimization level, as this would be normally the standard setting one chooses to get a high degree of optimization.

Setting	Command-line Option for GCC
O0	-O0
O1	-O1
O2	-O2
O3	-O3
FULL	-O3 –unroll-loops –unroll-all-loops
SCCP1	-O3 –sccp-enforce –sccp=sc
SCCP2	-O3 –sccp-enforce –sccp=sc,cc,dc,mcdc,pc

Table 2. Compiler Options for the Different Settings

The numerical results of the performance comparison with different optimizations and preservation of coverage metrics is shown in Table 3. What we actually see from these first results is, that the compiler operation with any SCCP mode is nearly always very close or equal to the optimization mode `-O3`. This is a very good result as it means that with GCC 4.5.2 ensuring preservation of structural code coverage does not cost any significant performance, but provides an extra confidence in the coverage of a test suite obtained at the source-code level.

Table 3 shows also some interesting performance results of GCC in general. For example, in case

Name	#LOC	Measured Execution Time (ms)						
		O0	O1	O2	O3	FULL	SCCP1	SCCP2
qurt	166	45.00	28.00	29.00	28.00	24.00	28.00	28.00
adpcm	878	99.00	55.00	68.00	65.00	64.00	65.00	65.00
matmult	177	52.00	24.00	22.00	15.00	13.00	16.00	15.00
ludcmp	147	111.00	45.00	42.00	34.00	39.00	35.00	34.00
jfdctint	375	77.00	35.00	34.00	31.00	31.00	31.00	31.00
crc	128	31.00	19.00	20.00	18.00	18.00	18.00	18.00
edn	285	128.00	43.00	42.00	28.00	29.00	28.00	28.00

Table 3. Performance Evaluation of SCCP

of benchmark *adpcm* we see that the performance of the generated code is significantly better with optimization level `-O1` than with `-O3`. This is a quite common phenomenon, as code optimizations also have side effects like increasing code size or changing memory layout, which in some cases can turn down any performance gain. This is also the reason why many other code optimizations are not included automatically when choosing `-O3`. From the SCCP method this is a neutral result, because if the SCCP settings can match the `-O3` setting, it implies that in can also handle the `-O1` setting with the same precision as there are less optimizations included in `-O1`. We also see there that the optimization setting FULL does not always give a better result than O3 does, as can be seen for benchmark *ludcmp*.

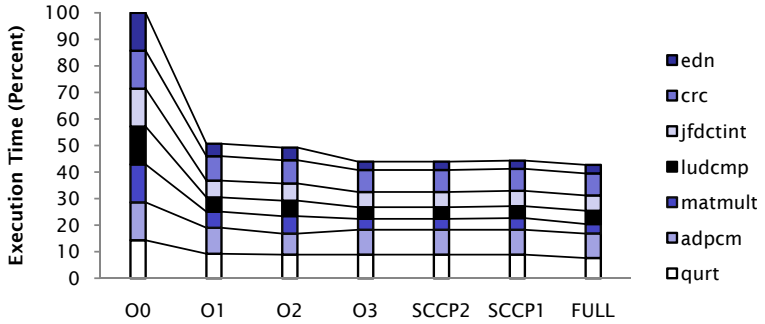

Figure 6. Performance of the Different Optimizations, subsumed for a Number of Benchmarks

Figure 6 subsumes the experimental results as a combined graph for all benchmarks. Each vertical bar shows the total execution time of all the benchmarks summed up for one optimization setting. Here we can see directly, that the average performance of both SCCP settings matches that of `-O3`.

To summarize, the performance results of integrating the SCCP approach into GCC has proven to be done quite easily and the performance is quite the same as without SCCP.

6. Summary and Conclusion

In this paper we have described the issue of tracing structured code coverage from the source code to the machine code in case of optimizing compilation. We have described the need for portable test data generation for our MBTA tool FORTAS, which uses a combination of different test data generation methods to achieve efficient generation and high coverage. However, there are code optimizations that can introduce additional control-flow decisions to improve the average-case performance, but increase the WCET. When test data are generated from the source code, it is quite likely that such additional control-flow decisions are not coverage by the test suite, providing a cause for WCET underestimation.

We have described a solution to this problem, by developing a coverage preservation mode for the compiler, which guarantees that in case a concrete structural code coverage has been achieved at the original code, it will also be fulfilled at the transformed code. To do this, we infer from the formalization of the code transformations and the coverage-preservation criteria whether the code transformation guarantees preservation of the structural code coverage criterion. We demonstrated the light effort necessary to integrate this approach into a compiler, by providing a prototype implementation for GCC 4.5.2. Experimental results have shown that this technique provides quite negligible performance costs, which allows it to use for timing analysis for final production code.

References

[1] Powertrace. Product Information from Lauterbach GmbH, Höhenkirchen-Siegertsbrunn, Germany.

[2] BETTS, A., MERRIAM, N., AND BERNAT, G. Hybrid measurement-based wcet analysis at the source level using object-level traces. In *Proc. 10th International Workshop on Worst-Case Execution Time Analysis* (Brussels, Belgium, July 2010), pp. 54–63. Available: `http://drops.dagstuhl.de/opus/volltexte/2010/2825`.

[3] BÜNTE, S., ZOLDA, M., AND KIRNER, R. Let's get less optimistic in measurement-based timing analysis. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES'11)* (June 2011).

[4] BÜNTE, S., ZOLDA, M., TAUTSCHNIG, M., AND KIRNER, R. Improving the confidence in measurement-based timing analysis. In *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)* (Mar. 2011).

[5] HOLZER, A., TAUTSCHNIG, M., VEITH, H., AND SCHALLHART, C. How did you specify your test suite? In *25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (Sept. 2010), pp. 407–416.

[6] KIRNER, R. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems 2009* (2009).

[7] KIRNER, R., AND HAAS, W. Automatic calculation of coverage profiles for coverage-based testing. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung* (Maria Taferl, Austria, Oct. 2009).

[8] Mälardalen research and technology centre WCET benchmarks. Web page (`http://www.mrtc.mdh.se/projects/wcet/`), 2009. Accessed online in April 2011.

[9] STALLMAN, R. M., AND GCC DEVELOPER COMMUNITY. *GNU Compiler Collection Internals*. GNU Press, Boston, USA, Dec. 2010. GCC version 4.5.2, available online at `http://gcc.gnu.org/gcc-4.5/`.

[10] STALLMAN, R. M., AND GCC DEVELOPER COMMUNITY. *Using the GNU Compiler Collection*. GNU Press, Boston, USA, Dec. 2010. GCC version 4.5.2, available online at `http://gcc.gnu.org/gcc-4.5/`.

[11] VIENNA UNIVERSITY OF TECHNOLOGY. The SECCO project: Sustaining entire code-coverage on code optimization. web page (`http://pan.vmars.tuwien.ac.at/secco/`). accessed in May 2011.

[12] WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. Measurement-based timing analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Porto Sani, Greece, Oct. 2008).

[13] ZOLDA, M., BÜNTE, S., AND KIRNER, R. Context-sensitivity in IPET for measurement-based timing analysis. In *Proc. 4th Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Oct. 2010), Springer Verlag.

[14] ZOLDA, M., BÜNTE, S., AND KIRNER, R. Context-sensitive measurement-based worst-case execution time estimation. In *17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)* (Aug. 2011). submitted.