SCCP/x - A Compilation Profile to Support Testing and Verification of Optimized Code [*]

Raimund Kirner
Institut für Technische Informatik
Technische Universität Wien, Austria
raimund@vmars.tuwien.ac.at

ABSTRACT

Embedded systems are often used in safety-critical environments. Thus, thorough testing of them is mandatory. A quite active research area is the automatic test-case generation for testing embedded systems. To achieve high retargetability of the testing framework, the test-case generation has to be done at source-code level. However, it is challenging to guarantee that the test-cases obtained from the source code are also valid at the object-code level, since even in safety-critical domains programs are optimized during compilation, i.e., the compiler may also introduce new control-flow decisions to the program.

In this paper we address the issue of how to guarantee the preservation of structural code coverage of test data during the optimizing compilation of the program. We analyze the formal program properties that have to be maintained to preserve different structural testing coverages, like *branch coverage* or *modified condition/decision coverage*. Based on this we describe a compilation profile that can be integrated into a compiler to allow the enforcement of structural code-coverage preservation. This work was motivated by current research activities to generate test data automatically from the source code, for example, for measurement-based timing analysis of real-time programs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Theory

[*]This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (COSTA) under contract P18925-N13 and by the ARTIST2 Network of Excellence (`http://www.artist-embedded.org/`)

1. INTRODUCTION

In embedded computing, testing is very important, since the system behavior typically depends on the target platform, like, for example, the correct timing behavior of the system. In case of safety-critical computing, there are additional requirements to be met. For example, the RTCA/DO178b, which is a document describing software engineering for commercial airborne systems, requires that structural code coverage analysis has to be used as an additional measure to ensure adequate testing [12].

Using structural code coverage as an additional measure for functional testing is a well-known approach. Sometimes, structural code coverage was mistakenly referred to as primary goal in functional testing for generating test data (see the clarification in [6]). However, there are also legitimate testing approaches using structural code coverage as primary metrics for automatic generation of test data. For example, in measurement-based worst-case execution time (WCET) analysis one is interested in achieving custom structural code coverage of execution time measurements [14, 13, 15, 2].

Besides the mature research on applying structural code coverage, there doesn't seem to be any research on how to preserve the structural code coverage when modifying the code. Actually, this is the subject of this paper, because we want to automatically generate test data from the source code, since this allows easy retargeting of the framework to new platforms. But at the same time we have to ensure that no hidden paths in the object code have been missed when we are doing measurement-based WCET analysis.

To describe the effects of code transformations, we developed notations described in Section 2. To analyze the effects of structural code coverage criteria, we formally model them in Section 2.1. Based on the formal descriptions of structural code coverage criteria, we develop in Section 3 restrictions that code transformations must fulfill to preserve code coverage.

In Section 4 we show how to integrate this compilation profile into a compiler and give a classification of several code transformations. Besides conventional compilers, this compilation profile also applicable to source-to-source transformations.

2. BASIC DEFINITIONS

In this section we give a list of definitions that will be used in the following sections to describe the properties of structural code coverage and how to preserve them.

basic block of a program P is a code sequence of maximal length with a single entry point at the beginning and with the only allowed occurrence of a control-flow statement at its end. We denote the set of basic blocks in a program P_i as B_i.

decision is a Boolean expression composed of *conditions* that are combined by Boolean operators. If a *condition* occurs more than once in the *decision*, each occurrence is a distinct condition [3]. However, the *input* of a condition is the set of its conditions without duplicates. We denote the set of decisions of a program P_i as D_i.

Note, that even in the C statement `a = (b && c);` the expression `(b && c)` is a decision. This makes sense, since due to the short-circuit evaluation semantics of the `&&` operater, the compiler will produce for this expression code with conditional control flow.

condition is a Boolean expression containing no Boolean operators [3]. We denote the set of conditions of a decision d as $\mathrm{C}(d)$. The set of all conditions within a program P_i is denoted as $\mathrm{C}(D_i)$.

P_1**,** P_2 denotes the program P before (P_1) and after (P_2) the transformations for which we want to preserve structural code coverage.

test data *TD* is the set of test data that has been generated with structural code coverage analysis done at source-code level. *TD* is a (true) subset of the program's input data space *ID*.

reachability valuation $IV_R(x)$ defines the set of valuations[1] of the input variables that trigger the execution of expression x.

satisfiability valuation $IV_T(x)$**,**$IV_F(x)$ defines the sets of valuations of the input variables that trigger the execution of the Boolean expression x with a certain result of x: $IV_T(x)$ is the input-data set where x evaluates to True and $IV_F(x)$ is the set where x evaluates to False. The following properties always hold for $IV_T(x)$,$IV_F(x)$:

$$IV_T(x) \cap IV_F(x) = \emptyset$$
$$IV_T(x) \cup IV_F(x) = IV_R(x)$$

For example, consider the following C code:

```
void f (int a,b) {
   if (a==3 && b==2)
      return 1;
   return 0;
}
```

For this code fragment we have $IV_T(\texttt{b==2}) = \{\langle 3, 2\rangle\}$ (and not the larger set $\{\langle a, 2\rangle \mid a \in \texttt{int}\}$). Only those input data that trigger the execution of condition `b==2` and evaluate it to True are within $IV_T(\texttt{b==2})$. Further, is holds that $IV_F(\texttt{b==2}) = \{\langle 3, b\rangle \mid b \in \texttt{int} \wedge b \neq 2\}$.

Above definitions allow to formally describe structural code coverage criteria formally. We will also use them to describe requirements to preserve structural code coverage.

[1]Valuation of a variable means the assignment of concrete values to it. The valuation of an expression means the assignment of concrete values to all variables within the expression.

2.1 Structural Code-Coverage Criteria

Structural code-coverage criteria measure the flow of control between a program's statements. The satisfaction of a structural code-coverage criteria isn't the primary test-case generation strategy in functional testing. Instead, structural code-coverage achieved during testing is analyzed as a supplementary measure to decide whether the implemented functionality has been sufficiently tested and does not contain any unintended functionality. However, there are also applications of testing where the satisfaction of a certain code-coverage is also the primary directive for test-data generation. For example, in measurement-based timing analysis an estimation of the worst-case execution time (WCET) is derived by systematic measurements [13].

In the following we review the properties of several structural code-coverage criteria:

Line coverage: is not a serious code coverage criteria, as without strict code guidelines it is not defined what a source line consists of. Historically, line coverage was used as an easy hack, where tools for analyzing *statement coverage* were missing. Thus, we do not discuss preservation of line coverage in this work.

Statement coverage (SC) : requires that every statement of the program is executed at least once. Statement coverage alone is quite weak for functional testing [10] and should best be considered as a minimal requirement. Using our above definitions, we can formally define SC as follows:

$$\forall b \in B.\ (TD \cap IV_R(b)) \neq \emptyset \tag{1}$$

Decision coverage (DC): requires that each *decision* of the program has been tested at least once with each possible outcome. Decision coverage is also known as *branch coverage* or *edge coverage*. Decision coverage implies *statement coverage*.

$$\forall d \in D.\ (IV_T(d) \cap TD) \neq \emptyset \wedge (IV_F(d) \cap TD) \neq \emptyset \tag{2}$$

Condition coverage (CC): requires that each *condition* of the program has been tested at least once with each possible outcome. It is important to mention that CC does *not* imply DC. A formal definition of CC is given in Equation 3

$$\forall c \in \mathrm{C}(D).\ (IV_T(c) \cap TD) \neq \emptyset \wedge (IV_F(c) \cap TD) \neq \emptyset \tag{3}$$

Note that our definition requires in case of short-circuit operators that each condition is really executed. This is achieved by the semantics of $IV_T()$,$IV_F()$. However, often definitions are used that do not explicitly consider short-circuit operators (as, for example in [6]), thus having in case of short-circut operators only a "virtual" coverage since they do not guarantee that the short-circuit condition is really executed for the evaluation to True as well as for the evaluation to False.

Condition/Decision coverage (CDC): requires that both, *condition coverage* and *decision coverage* are achieved.

Modified Condition/Decision Coverage (MC/DC): requires to show that each condition can independently affect the outcome of the decision [12]. Thus, having n conditions in a decision, $n+1$ test cases are required to achieve MC/DC. Note, that MC/DC also implies both, DC and CC. A formal definition of MC/DC given in [8].

The original definition of MC/DC given in the RTCA/DO178b document [12] is rather strict, so that people thought of some less restrictive definitions. For example, it is not possible with the original defintion to cover a decision with strongly coupled conditions.[2] As described in [3], there exist at least three definitions of MC/DC:

- *Unique-Cause MC/DC*: this is the original definition given in [12].
- *Unique-Cause + Masking MC/DC*: this definition of MC/DC is less restrictive as it requires in case of strongly coupled conditions to test only that one of them covers the decision (masking) [4].
- *Masking MC/DC*: this definition is less restrictive than the two above, as it does not require anymore to test whether conditions do independently cover the decision. It focuses more on testing the correct implementation of subexpressions within a Boolean expression.

For sake of simplicity, we focus within this work on the original definition of MC/DC. However, extending the formal definition of MC/DC to *Unique-Cause + Masking MC/DC* is straight-forward, as shown above.

3. PRESERVATION OF CODE COVERAGE

When transforming a program, we are interested in the program properties that must be maintained by the code transformation such that a structural code coverage of the original program by the test-data set *TD* is preserved to the transformed program. Based on this properties one can adjust a src-to-src transformer or a compiler to use only those optimizations that preserve the intended structural code coverage.

$$\begin{aligned} &\mathit{is\_bool\_covered}(x', x_T, x_F) \Rightarrow \\ &\quad \exists S_1, S_2 \subseteq ID.\ \mathit{is\_TFSubset}(S_1, S_2, x') \wedge \\ &\qquad (S_1 \supseteq IV_T(x_T) \wedge S_2 \supseteq IV_F(x_F)); \end{aligned} \tag{4}$$

$$\begin{aligned} &\mathit{is\_TFSubset}(S_1, S_2, x) \Rightarrow \\ &\qquad (S_1{=}IV_T(x)\ \vee\ S_1{=}IV_F(x)) \wedge \\ &\qquad (S_2 = (IV_T(x) \cup IV_F(x)) \setminus S_1); \end{aligned}$$

The predicate symbol $\mathit{is\_bool\_covered}(x', x_T, x_F)$ (defined in Equation 4) tests whether a Boolean coverage (covering the result values TRUE and FALSE) of the expressions x_T,x_F implies that also the expression x' is covered. The auxiliary symbol $\mathit{is\_TFSubset}(S_1, S_2, x)$ states that one of the input-data subsets S_1,S_2 matches the input data set $IV_T(x)$ and the other one matches the input data set $IV_F(x)$. The symbol $\mathit{is\_bool\_covered}(x', x_T, x_F)$ is used in the following on the discussion of coverage preservation of the different coverage metrics.

[2]Two conditions c_1,c_2 are strongly coupled, iff $IV_T(c_1) = IV_T(c_2)$

In the following subsections we show the program properties that have to be maintained by the code transformations to preserve the different structural code coverage criteria. The preservation of the MC/DC code coverage is described in [8].

3.1 Statement Coverage (SC)

THEOREM 3.1. **(Preservation of SC)** *Assuming that a set of test data TD achieves* statement coverage *on a given program P_1, then Equation 5 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of statement coverage on a transformed program P_2.*
(Proof given in [8])

$$\forall b'{\in}B_2\ \exists b{\in}B_1.\ IV_R(b') \supseteq IV_R(b) \tag{5}$$

3.2 Condition Coverage (CC)

THEOREM 3.2. **(Preservation of CC)** *Assuming that a set of test data TD achieves* condition coverage *on a given program P_1, then Equation 6 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of condition coverage on a transformed program P_2.*

$$\forall c'{\in}\mathrm{C}(D_2)\ \exists c_1, c_2{\in}\mathrm{C}(D_1).\ \mathit{is\_bool\_covered}(c', c_1, c_2) \tag{6}$$

Proof (Preservation of CC): Part 1, showing sufficiency: Since *TD* is assumed to achieve CC on P_1, it holds for each $c \in \mathrm{C}(D_1)$ that for each $c' \in \mathrm{C}(D_2)$ $(IV_T(c) \cap TD) \neq \emptyset$ and $(IV_F(c) \cap TD) \neq \emptyset$. Since Equation 6 states that

$$\exists c{\in}\mathrm{C}(D_1).\ (IV_T(c') \supseteq IV_T(c) \vee IV_T(c') \supseteq IV_F(c)) \text{ and}$$
$$\exists c{\in}\mathrm{C}(D_1).\ (IV_F(c') \supseteq IV_F(c) \vee IV_F(c') \supseteq IV_T(c)), \text{ it}$$

follows that for each $c' \in \mathrm{C}(D_2)$ we also have $(IV_T(c') \cap TD) \neq \emptyset$ and $(IV_F(c') \cap TD) \neq \emptyset$. Thus, CC is preserved at P_2.
Part 2, showing necessity by indirect proof: Assuming there exists a condition $c' \in \mathrm{C}(D_2)$ of P_2 such that for all conditions $c_1, c_2 \in \mathrm{C}(D_1)$ of P_1 it either holds that
a) $\neg(IV_T(c') \supseteq IV_T(c_1) \vee IV_T(c') \supseteq IV_F(c_1))$, and
b) $\neg(IV_F(c') \supseteq IV_F(c_2) \vee IV_F(c') \supseteq IV_T(c_2))$, then
it can happen that
a) $\forall c_1{\in}\mathrm{C}(D1).\ TD \cap IV_T(c') \cap (IV_T(c_1) \cup IV_F(c_2)) = \emptyset$,
or
b) $\forall c_2{\in}\mathrm{C}(D1).\ TD \cap IV_F(c') \cap (IV_F(c_2) \cup IV_T(c_2)) = \emptyset$,
which in both cases violates the preservation of CC. □

3.3 Decision Coverage (DC)

THEOREM 3.3. **(Preservation of DC)** *Assuming that a set of test data TD achieves* decision coverage *on a given program P_1, then Equation 7 provides a sufficient - and without further knowledge about the program and the test data, also necessary - criterion for guaranteeing preservation of decision coverage on a transformed program P_2.*
(Proof given in [8])

$$\forall d'{\in}D_2\ \exists d_T{\in}D_1\ \exists d_F{\in}D_1.\ \mathit{is\_bool\_covered}(d', d_T, d_F) \tag{7}$$

4. INTEGRATION INTO COMPILER

The final goal of this research would be to have a compiler where the user can select compilation profiles that ensure that structural code coverage criteria are preserved during optimizing compilation. For example, the compiler might be started with a command line option like

```
cc --SCCP/DC file.c
```

to enable *structural code coverage preservation* of *decision coverage* (DC). Of course, the suffix "DC" may be replaced by any other code coverage criteria of which the compiler is able to preserve coverage. The compiler will be restricted to use only those code transformations that do not disrupt a given structural code coverage.

The formal criteria that must be fulfilled to preserve a given structural code coverage criteria are presented in Section 3. The interesting question is, how to use these criteria to decide on a series of code transformations whether they preserve the interested structural code coverage.

One possible approach would be to formalize each code transformation in an axiomatic semantics [5, 7], i.e., by providing preconditions, postconditions and invariants of the code transformation. A research area quite related to this is the *translation validation* of optimizing compilers [11, 16]. Within this work, we analyze the transformations manually. Once, a classification of code transformations is available, including this profile into a compiler shouldn't be that much effort. Thus, we hope that also commercial compiler vendors will adopt the idea and integrate it into their compilers.

4.1 Classification of Code Transformations

There are two types of code transformations that challenge the preservation of structural code coverage:

- transformations that change the reachability of statements or conditions. For example, reordering the conditions within a decision can change the reachability of conditions and thus also statements.
- transformations that add new conditional control-flow paths into the program. For example, branch optimization can introduce new conditional branches.

Sometimes, the effects of code transformations are not quite obvious. For example, even *common subexpression elimination* (aka CSE) can modify the control-flow paths, since a subexpression can also contain conditional control flow, as for example, the C operators `&&` and `||` do.

A summary of several code transformations used in optimizing compilers is given in Table 1. The column *Transformation* lists the name of different code transformations. We do not describe these transformations here; one can find detailed descriptions of them in [1, 9]. The column *Cat* describes how the control flow is changed: control flow is not changed (=), paths are removed ($\perp$), paths are added (+), or paths are modified ($\leftrightarrows$). Paths get added by introducing new conditions, or splitting existing ones. Paths get removed by removing or joining existing conditions. We say that paths get modified, if the criteria for both, adding and removing do apply. The four right-most columns states whether the code transformation preserves the given structural code coverage (√) or not (-). Classifications given in parenthesis mean that there is a different effect between typical implementations and the generic form of a code transformation.

Transformation	*Cat*	*SC*	*CC*	*DC*	*MC/DC*
Algebraic simplif.	(=)	(√)	(√)	(√)	(√)
CSE	$\leftrightarrows$	√	√	√	√
Constant folding	=	√	√	√	√
Branch optimization	$\leftrightarrows$	-	-	-	-
Empty loop removal	$\perp$	√	√	√	√
Loop blocking	$\leftrightarrows$	-	-	-	-
Loop fusion	$\leftrightarrows$	√	√	√	√
Loop interchange	$\leftrightarrows$	-	-	-	-
Loop inversion	+	√	-	-	-
Loop peeling	$\leftrightarrows$	-	-	-	-
Loop unrolling	$\leftrightarrows$	-	-	-	-
Software pipelining	($\leftrightarrows$)	-	-	-	-
Unreachable code elimination	$\perp$	√	√	√	√
Useless code elimin.	$\perp$	√	√	√	√

Table 1: Classification of Code Transformations

Algebraic simplification preserves the coverage criteria in case of common implementations that do not introduce new operators with conditional control flow. However, the categorizations are given in parenthesis, since special implementations may also introduce new operators that contain contain conditional control flow that split the set $IV_R()$ of the expression into two sets $IV_T()$ and $IV_F()$, in which case no preservation of code coverage is possible. *CSE* might move expressions with conditional control flow, however their coverage is preserved, as for their conditions c the sets $IV_R(c)$, $IV_T(c)$, and $IV_F(c)$ are not changed. *Constant folding* preserves code coverage because it does not shrink the sets $IV_R()$, $IV_T()$, and $IV_F()$ of any decision or condition.

Branch optimization does not guarantee code coverage as it can shrink the sets $IV_R()$, $IV_T()$, and $IV_F()$ of decisions or conditions.

Loop fusion can introduce new paths because the bodies of the fused loops get merged. However, as both original loops have the same iteration count, the sets $IV_R()$, $IV_T()$, and $IV_F()$ of decisions or conditions inside the loop bodies do not change, thus preserving all code coverages.

Loop inversion splits the loop exit decision d into two separate decisions, i.e., it splits the sets $IV_R(d)$, $IV_T(d)$, and $IV_F(d)$, thus all coverages except SC are not preserved. SC is preserved, because the set $IV_R()$ of the loop body may not be changed.

Loop blocking, *loop interchange*, *loop peeling*, *software pipelining*, and *loop unrolling* do not preserve any of the mentioned code coverage, as none of the coverage criteria guarantees that all paths within the original loop are covered, as they may, for example, split the sets $IV_R()$, $IV_T()$, and $IV_F()$ of decisions or conditions.

Empty loop removal, *unreachable code elimination*, and *useless code elimination* may only reduce the number of paths, in which case the sets $IV_R()$, $IV_T()$, and $IV_F()$ of decisions or conditions may only increase; thus, all mentioned code-coverage metrics are preserved.

As a final note, one should take the results of this table with care, since every compiler vendor may implement transformations with the same name slightly different.

4.2 Assembly-Code Generation

There is a general difference between source-to-source program transformations and assembly code generation. Doing source-to-source transformations, the destination language still provides the same features as the target language to code control-flow decisions composed of multiple conditions. However, when generating assembly code, it happens that the instruction set of most processors does not include instructions with complex control-flow decisions. Thus, numerous operations are used to set Boolean flags and a conditional jump evaluates these flags to decide whether to transfer control to the jump target or not. Thus, at assembly code there is typically no distinction between decision and condition, as all decisions contail only one condition.

As a consequence, at assembly level there is no difference between *decision coverage* (*branch coverage*), *condition coverage*, and MC/DC. To give an example of how generated assembly code would look like, consider the following simple C program code:

```
if (a && (b || c)) { ... }
```

An example of a possible assembly code structure for this code would be

```
if (!a) goto skip;
if ( b) goto process;
if (!c) goto skip;
process:
  ...
skip:
```

In the above example, every condition has become its own decision, though sometimes being inverted. And as long as the compiler uses such a code generation schema, it holds on assembly level that *branch coverage* is the same as CC, DC, or MC/DC.

5. SUMMARY AND CONCLUSION

Preserving structural code coverage seems to be an almost new field of research. Our motivation is to ensure that measurement-based worst-case execution time (WCET) analysis based on test data derived from the source code really covers the intended execution paths, and that there are no code structures missed by the tests. Current practice is to analyze the coverage at object-code level, requiring additional analysis tools that support the concrete platform.

In this paper we worked out the fundamentals of how to achieve preservation of structural code coverage when transforming programs. While our original motivation was to achieve solely *decision coverage* on assembly code, we analyzed four different structural code coverage criteria and developed criteria of how to preserve them: *statement coverage* (SC), *decision coverage* (DC), *condition coverage* (CC) and *modified condition/decision coverage* (MC/DC).

First classifications of code transformations, done manually based on the formal criteria, have shown that the criteria are helpful to decide whether a code transformation preserves a given structural code coverage. Future work would be to apply this classification in a more formal way, e.g., by formalizing the transformations themself and using formal verification to decide whether the preservation of coverage is achieved. Further, we plan to implement this approach in an open source compiler and perform a quantitative analysis of the impact on the WCET when restricting the allowed set of code transformations to preserve a specific code coverage.

6. REFERENCES

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[2] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[3] J. J. Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical Report DOT/FAA/AR-01/18, Boeing Commercial Airplane Group, Apr. 2001.

[4] J. J. Chilenski and S. Miller. Applicability of modified condition decision coverage to software testing. *Software Engineering Journal*, 7(5), Sep. 1994.

[5] R. Floyd. Assigning meaning to programs. In *Proc. of AMS Symposia in Applied Mathematics*, pages 19–32, 1967.

[6] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutoral on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, National Aeronautics and Space Administration, Hampton, Virginia, May 2001. available in pdf format.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[8] R. Kirner. Formal requirements for structural code-coverage preservation of code optimization: The SCCP/x framework. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2007.

[9] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

[10] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[11] G. C. Necula. Translation validation for an optimizing compiler. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95, Vancouver, 2000.

[12] Software considerations in airborne systems and equipment certification. RTCA/DO-178B, 1992.

[13] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proc. Conference on Design, Automation and Test in Europe (DATE'05)*, pages 606–611, Munich, Germany, Mar. 2005. IEEE.

[14] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Measurement-based worst-case execution time analysis. In *Proc. 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10, Seattle, Washington, May 2005.

[15] F. Wolf, J. Kruse, and R. Ernst. Segment-wise timing and power measurement in software emulation. In *Proc. IEEE/ACM Design, Automation and Test in Europe Conference, Designers' Forum*, pages 165–169, Munich, Germany, Mar. 2001.

[16] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A translation validator for optimizing compilers. In *International workshop on Compiler Optimization meets Compiler Verificaiton (COCV)*, pages 178–190, Apr. 2002.