Improving the Confidence in Measurement-Based Timing Analysis

Sven Bünte, Michael Zolda, Michael Tautschnig
Vienna University of Technology
{sven,michaelz}@vmars.tuwien.ac.at
tautschnig@forsyte.at

Raimund Kirner
University of Hertfordshire
r.kirner@herts.ac.uk

***Abstract*—Measurement-based timing analysis (MBTA) is a hybrid approach that combines execution-time measurements with static program analysis techniques to obtain an estimate of the worst-case execution time (WCET) of a program. The most challenging part of MBTA is test data generation. Choosing an adequate set of test vectors determines safety and efficiency of the overall analysis. So far, there are no feasible criteria that determine how well the worst-case temporal behavior of program parts is covered by a given test-suite.**

In this paper we introduce a relative safety metric that compares test suites with respect to how well the observed worst-case behavior of program parts is exercised. Using this metric, we empirically show that common code coverage criteria from the domain of functional testing can produce unsafe WCET estimates in the context of MBTA for systems with a processor like the TriCore 1796. Further, we use the relative safety metric to examine coverage criteria that require all feasible pairs of, e.g., basic blocks to be exercised in combination. These are shown to be superior to code coverage criteria from the domain of functional testing, but there is still a chance that an unsafe WCET estimate is derived by MBTA in our experimental setup. Based on the outcomes of our evaluation we introduce and examine *Balanced Path Generation*, an input data generation technique that combines the advantages of all evaluated coverage criteria and random input data generation.

***Index Terms*—Real-time systems, validation, worst-case execution time, structural code coverage**

I. Introduction

A real-time computer system is a computer system in which correctness does not only encompass functional behavior, but also compliance to temporal constraints. If the violation of the timing constraints can have catastrophic consequences we speak of a hard real-time system. Otherwise, it is called soft real-time system. An example of the latter is a mobile phone application that can tolerate minor communication delays. As an example for hard real-time systems, an airbag not releasing in time or a non-reacting aircraft control unit can lead to catastrophic consequences. Consequently, there is an inherent interest in verification and validation techniques that focus on the temporal behavior of real-time systems.

Many real-time computer systems are implemented as a collection of individual tasks in order to handle complexity. A valid schedule for those tasks ensures the adherence to temporal dependencies [1]. Most of the common scheduling algorithms rely on the *Worst-Case Execution Time* (WCET) of each single task.

Determining the WCET of a program by simple end-to-end measurements exercising the program with different input data is unlikely to find a safe upper bound on the real WCET, i.e., a bound that is never exceeded under any circumstances. As a more systematic approach that provides much higher confidence into safety of the obtained WCET bound, static WCET analysis is based on the provision of an accurate timing model of the processor [2]. Given that the involved analysis techniques are sound and that the timing model is correct, static WCET analysis can yield a safe upper bound on the WCET. However, timing properties of modern architecture features like caches, branch predictors or out-of-order execution are hard to analyze precisely. As a consequence, static analysis struggles to deliver *precise* results, i.e., WCET bounds that are only slightly larger than the real WCET, due to conservative assumptions it has to act on in order to be safe. Further, the manual construction of a detailed timing model is often only economically feasible for processors used in safety-critical systems where comprehensive verification efforts are mandatory for certification.

Thus, measurement-based timing analysis has emerged, where the timing model is obtained from execution-time measurements [3].

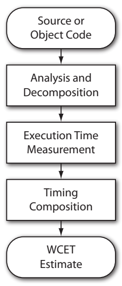

Fig. 1. The three phases of measurement-based timing analysis (MBTA)

A. *Measurement-Based Timing Analysis*

Measurement-based timing analysis (MBTA) is a hybrid WCET analysis technique. It combines static program analysis techniques and execution time measurements. As shown in Figure 1, measurement-based timing analysis typically consists of the three phases *Analysis and Decomposition*, *Execution Time Measurement*, and *Timing Composition*.

- **Analysis and Decomposition:** For WCET analysis, the maximal end-to-end execution time of the software is of interest. In general, to obtain a perfectly accurate timing model, we would have to consider the execution time of all possible operation sequences that can be performed by the computer for all possible initial states of the system under scrutiny while executing the given computer program. Measuring all these sequences is intractable in general, as there are simply too many. Therefore, reducing the number of execution time measurements is crucial. Usually, MBTA approaches try to access the local WCET of program segments (subgraphs [4]–[6], sequences between instruction points [7], down to basic blocks [4]). All these decomposition techniques of execution-time measurements imply heuristics for selecting representative test data for covering the worst-case timing effects of the program. The effectiveness of such a heuristic depends on the nature of the target hardware, e.g., for processors with caches we also need to consider global effects.
  However, there are also approaches that estimate the global WCET of a program without decomposing it by considering end-to-end execution times using evolutionary algorithms [8], [9].
- **Execution Time Measurement:** Once the program is decomposed, the execution time is estimated for each segment. Execution times are measured on real hardware, which allows us to take hardware characteristics into account without modeling them in full detail. In general, the MBTA approach does not provide sufficient coverage of all system states to guarantee that the maximal observed execution time (MOET) for a segment is indeed the WCET. This is due to hardware features like pipelines, caches, and out-of-order execution that blow up the state space of the processor. Further, all possible system states at the entry of a code fragment would have to be covered in order to guarantee that the WCET is among the MOETs. Thus, we strongly advocate coverage criteria that aim for exercising the worst-case temporal behavior of program segments by considering the aforementioned hardware effects.
- **Timing Composition:** The timing results from all segments are composed via IPET [10], [11] or a tree-based approach [7], [12] to obtain a global WCET estimate (we call a WCET estimate safe, if it does not fall below the real WCET).

In contrast to static WCET analysis, MBTA does not try to guarantee safety for all of its results. It rather aims for balancing analysis efforts to serve both needs: to yield safe results on the one hand and to be precise on the other hand. Obviously, compromises have to be made. With respect to the verification of temporal requirements in embedded systems, this implies that MBTA targets soft real-time systems primarily, where precision of a calculated WCET bound is as crucial as safety. However, MBTA can also be used to verify hardware models used in static WCET analysis or for design space exploration of both hard and soft real-time systems.

This article will focus on the aspect of safety in MBTA, while a survey of the competing aspect of precision is provided in [13]. Recall that MBTA involves test data generation as an essential step in the *Execution Time Measurement* phase. So far, no coverage metric has been shown to adequately determine if a code segment is exercised sufficiently by a given test suite for obtaining an adequately safe WCET estimate. Recall that the main benefit of MBTA is that is does not require a hardware model of the system under scrutiny, which leads us to a dilemma: to guarantee that a coverage metric does imply the coverage of execution scenarios that can be used to obtain an upper bound of the WCET it requires a sophisticated hardware model of the system.

Our first contribution is a relative safety metric that compares test suites with respect to how well the observed worst-case behavior of program parts is exercised. The metric only uses observations from measurements and does therefore not rely on a hardware model.

As a second contribution, given this metric to quantify relative safety, we empirically show that the following common code coverage criteria from the domain of functional testing can produce unsafe WCET estimates in the context of MBTA for systems with a processor like the TriCore 1796:

- Random Testing
- Basic Block Coverage [14]
- Condition/Decision Coverage [15]
- Modified Condition/Decision Coverage (MC/DC) [16]

Further we use the relative safety metric to examine coverage criteria that require all feasible pairs of, e.g., basic blocks to be exercised in combination. These are shown to be superior to code-coverage criteria from the domain of functional testing but still reveal safety insufficiencies in the context of MBTA.

The final contribution is the introduction and examination of *Balanced Path Generation*, an input data generation heuristic that tries to combine the advantages of all other evaluated coverage criteria while minimizing their disadvantages. For our experimental setup, *Balanced Path Generation* performed best in terms of minimizing the chance that a WCET estimate by MBTA is not safe enough.

Section II elaborates on the evaluated coverage criteria and software models. Section III discusses existing work on coverage criteria for MBTA. An empirical evaluation of the mentioned coverage criteria from the domain of functional testing is presented in Section IV as well as the corresponding examination of pair coverage criteria. Balanced Path Generation is introduced and evaluated in Section V, followed by a summary and conclusion in Section VI.

II. Preliminaries

In order to express the coverage criteria under analysis we need a software model that is more detailed than the *Control Flow Graph (CFG)* with respect to control flow decisions. We therefore represent a program by means of a δ-*Control Flow*

Graph (δ-CFG) that combines a control flow graph with the statement-level precision of a control flow automaton [17]:

Definition II.1. A *δ-CFG* of a program $\mathcal{P}$ is a tuple $\langle G, v_0, \ell \rangle$ where $G = (V, E)$ is a directed graph with vertices V and edges $E \in V \times V$ with v_0 as the unique start of $\mathcal{P}$. Further, vertices and edges may be labeled by $\ell : S \subseteq (E \cup V) \rightarrow \{\texttt{T}, \texttt{F}\}$.

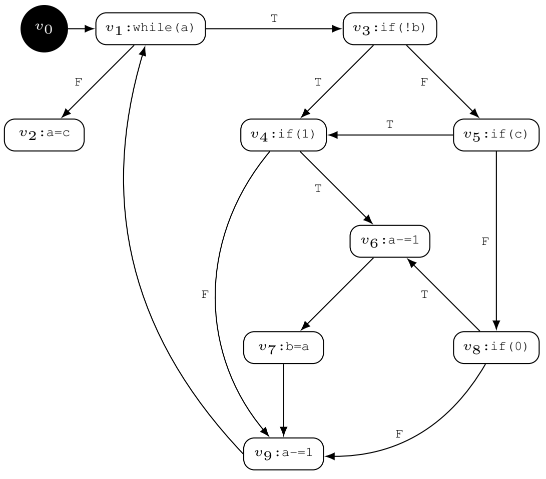

Fig. 2. δ-CFG for the example program in Listing 1

```
int a,b,c;
while(a){
   if(!b||c){
      a-=1;
      b=a;
   }
   a-=1;
}
a=c;
```

Listing 1. Example C program

Consider the example C program in Listing 1. A corresponding δ-CFG is illustrated in Figure 2. The main difference to the CFG is that short-circuit evaluation is made explicit by assigning one vertex to each condition of a boolean expression (v_1, v_3, v_5 are condition vertices in the example δ-CFG). Labels T and F refer to positive and negative outcomes of conditions respectively. Also, basic blocks (instruction sequences of maximal length where only the last instruction might be a jump) are not atomic entities in the δ-CFG: each program instruction is modeled by a single vertex.

A. Coverage Metrics

Although common coverage metrics such as condition coverage are widely used, they lack a standard definition and are interpreted distinctly by different tools. We shall therefore give a definition of those coverage metrics to be evaluated in this article. Before, we need some basic notations.

Definition II.2. Given a program $\mathcal{P}$, a *test vector* π assigns values to variables in $\mathcal{P}$. A *test suite* is a set Γ of test vectors. We say $\Gamma \models \gamma$ iff a test suite Γ *satisfies a coverage criterion* γ. Let $\Pi_{\mathcal{P}}(\pi)$ denote the δ-CFG path that results from an execution of $\mathcal{P}$ where all free variables are assigned with respect to π. We say vertex v is *exercised* by Γ iff there is a $\pi \in \Gamma$ such that v is in the path $\Pi_{\mathcal{P}}(\pi)$. v is *feasible* iff there exists a test vector π such that v is exercised. The maximal observed execution time for vertex v in the path $\Pi_{\mathcal{P}}(\pi)$ is denoted $t_{\mathcal{P}}(v, \pi)$). If v is not in $\Pi_{\mathcal{P}}(\pi)$, we set $t_{\mathcal{P}}(v, \pi)$ to -1.

Basic Block Coverage: Let each basic block v^i of a program $\mathcal{P}$ be expressed by a sequence of δ-CFG vertices $v^i = v^i_1, \ldots, v^i_k$. $\Gamma \models$ *Basic Block Coverage* iff for all v^i there is a j such that v^i_j is exercised by Γ. Basic block coverage is also referred to as line coverage or statement coverage [14]. Regarding the example program and its δ-CFG in Figure 2, it would for instance suffice to exercise v_0, v_2, v_6 and v_9. An example for a test suite satisfying basic block coverage is $\{\{(a, 2), (c, 1)\}\}$.

Decision Coverage: All feasible outgoing branches of control flow decisions are visited. Here, control flow decisions denote those in if-statements, while/for-statements, switch statements and conditional expressions ?:. Also referred to as branch coverage [15]. For our example, $\{\{(a, 2), (c, 1)\}, \{(a, 2), (b, 1), (c, 0)\}\}$ is a test suite that covers all decision outcomes (v_1, v_3), (v_1, v_2), (v_4, v_6), (v_8, v_9) and therefore satisfies decision coverage.

Condition Coverage: All feasible outcomes of all atomic conditions are exercised. This includes conditions in assignments such as a=a>2||c;. In our example, condition coverage requires (v_1, v_3), (v_1, v_2), (v_3, v_4), (v_3, v_5), (v_5, v_4), (v_5, v_8) to be exercised. Note that a test suite that satisfies condition coverage does not satisfy decision coverage in general. E.g. $\{\{(a, 2), (b, 0), (c, 0)\}$, $\{(a, 2), (b, 1), (c, 1)\}\}$ satisfies condition coverage but does not satisfy decision coverage in our example because !b||c never evaluates to F.

Condition/Decision Coverage: The union of condition coverage and decision coverage: all feasible outcomes of all atomic conditions and feasible outgoing branches of control flow decisions are exercised.

Modified Condition/Decision Coverage (MC/DC): According to [16], both of the following properties must hold:

- The respective test suite must guarantee condition/decision coverage.
- Each condition in a decision has been shown to independently affect that decision's outcome.

We consider the definition to target only feasible conditions. For our example program, $\{\{(a, 2), (b, 0), (c, 0)\}, \{(a, 2), (b, 0), (c, 1)\}, \{(a, 2), (b, 1), (c, 0)\}\}$ satisfies MC/DC coverage.

Pair Coverage: Given a coverage specification γ. Then $\gamma^2 = \gamma \times \gamma$ requires that all feasible pairs of instances in γ must be exercised. For example, (basic block coverage)^2 requires for a test suite to execute all basic blocks in combination at least once if the combination is feasible.

TABLE I
COVERAGE METRICS

Recall that given a program and a test suite, a coverage metric determines how well the program is tested/covered by the test suite. We will only examine complete test suites, i.e., those that fulfill 100% coverage. Hence, it is sufficient to define for each coverage metric those test suites that completely achieve the respective coverage. Table I provides definitions for the coverage criteria investigated in this article.

III. Related Work

The only research on coverage criteria for measurement-based timing analysis that we know of has been done by Betts et al. [18]. They propose three coverage metrics for processors with pipelines: *simple pipeline coverage*, *pairwise pipeline coverage*, and *hazard pipeline coverage*. The latter requires a processor-specific pipeline model and is therefore not in our scope. Both simple pipeline coverage and pairwise pipeline coverage are defined with respect to the instruction point graph [7] of a program whereas we use the δ-CFG. Apart from that, simple pipeline coverage is subsumed by condition/decision coverage and pairwise pipeline coverage is subsumed by both basic block/condition/decision pair coverage and extended basic block/condition/decision pair coverage. Further, we investigate the applicability (safety and feasibility) of the coverage criteria under scrutiny based on experiments where Betts et al. use explanatory examples. The major difference that distinguishes our work is that we do not restrict the analysis to execution time jitter due to structural and data hazards in pipelines. We compare the maximal observed execution times of program segments which can result from all hardware features the TriCore 1796 processor includes (branch prediction, instruction cache, pipeline effects, etc.). Colin et al. showed in [19] that considering the hardware of a system under scrutiny, caches have the greatest influence on a program's temporal behavior.

It is an interesting observation that the selection of adequate test vectors is crucial for MBTA on the one hand, but little studied on the other hand. We consider this fact as one more evidence that the dilemma of finding an appropriate coverage metric without modeling the hardware in detail is hard to solve. Our pragmatic approach (restricted to the TriCore processor and some benchmarks) should therefore be seen as a constructive step towards finding a metric that is applicable for most of the microprocessors and program types in the domain of embedded real-time systems.

IV. Evaluation of Common Coverage Criteria

As stated in the introduction, we do not want to model hardware features in MBTA. Consequently, we do not know the WCET of program segments and cannot give an absolute metric to determine if a segment is exercised sufficiently. Instead, we check how well a given coverage metric covers the worst-case behavior of program segments in relation to **all other** coverage metrics under investigation: for each δ-CFG vertex we identify the maximal observed execution time and calculate the differences to the maximal observed execution time considering all test suites for all coverage criteria. Then we sum up all differences to get the relative safety metric Φ.

Definition IV.1. Given a program $\mathcal{P}$ with n test suites such that $(\Gamma_1 \cup \ldots \cup \Gamma_n) = \Gamma_\cup$. The *relative safety* of test suite Γ_k is quantified by function Φ:

$$\Phi_{\mathcal{P}}(\Gamma_k) = \sum_{v \in V_{\mathcal{P}}} [\mathit{moet}\,(v, \Gamma_\cup) - \mathit{moet}\,(v, \Gamma_k)]$$

$$\mathit{moet}\,(v, \Gamma) = \max_{\pi \in \Gamma} [t_{\mathcal{P}}(v, \pi)]$$

Consequently, the lower the value of Φ, the better a program is covered in terms of exercising the worst-case execution times of the program's vertices **in relation to all test suites under analysis**. We have no means to reason about safety in an absolute way. For instance, if Φ is zero for a test suite Γ, this does not imply that the real worst-case execution time has been observed for any δ-CFG vertex. However, it shows that there is no test vector in any test suite under analysis that exercises a higher execution time, from which follows that test suite Γ is superior to all other test suites under analysis in terms of safety.

A. Experimental Setup

All measurements are performed on the TriCore 1796 microprocessor by Infineon. Programs under analysis are compiled with HighTec's GCC[1] compiler. The TriCore 1796 includes branch prediction, a superscalar pipeline and an instruction cache. Furthermore, it provides *On-Chip Debug Support (OCDS)* level 2 for cycle-accurate execution tracing. We utilize the Lauterbach LA-7690 Powertrace device to extract both timing and flow of control. Code instrumentation is thereby obsolete and measurements are cycle-accurate. For benchmarking our methods, we use the following programs in ANSI C:

- `bs` and `bsort`: Two implementations of the binary search and bubble sort algorithm, respectively, taken from the *Mälardalen WCET Project*[2]. The size of the input list for `bsort` is reduced from 100 to 10 as we utilize a bounded model checker (see below) that does not scale well for this particular benchmark.
- `lift_control`: The central control unit for an elevator that we translated to C as it is originally intended for the *Java Optimized Processor* [20]. The original version is used in the field and can be found on the web[3].
- An engine control unit from our industry partners. The code is generated by Matlab/Simulink and involves a more complex control flow structure than `lift_control`.

All benchmarks have been manually modified such that there is no short-circuit evaluation in boolean expressions, e.g., "`if(a&&b){`" is rewritten as "`if(a){if(b){`". Consequently, every decision is also an (atomic) condition after the translation process and any test suite that satisfies condition coverage also satisfies basic block coverage, condition/decision coverage and MC/DC for all benchmarks. We performed these modifications for two technical reasons: first, these changes ensure that the modeling and analysis tools we employ agree with our definition of a δ-CFG. Second, there is no doubt about the precise definitions of conditions, hence we are sure the formal semantics of FQL (see next paragraph) match our intuitive description in Table I.

All test suites (except for random input data generation) are specified by the *FShell Query Language (FQL)* [21] for our experiments. FQL is designed to specify test suites over source code and is expressive enough to describe the coverage metrics that we study in the following sections. Further, because FQL

[1]http://gcc.gnu.org/

[2]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[3]http://www.soc.tuwien.ac.at/trac/jop/browser/java/target/src/bench/jbe/lift

follows precise semantics, we can be sure that the specified test suites conform to the definitions in Section II.

To generate a test suite for a given specification in FQL we utilize FShell[4], a prototype implementation based on the principles described in [22], [23]. FShell relies on the C Bounded Model Checker (CBMC), version 3.8 [24]. The experiments are performed on a 2.66 GHz Intel Core2 Quad host equipped with 8 GB RAM.

Recall that Φ can quantify relative safety of test suites only. However, our goal is to evaluate safety of coverage criteria and input data generation techniques. Usually, there are many different test suites that satisfy a coverage criterion or that result from a generation process such as random input data generation. For illustration, imagine a test suite that satisfies coverage criterion Y and that yields a higher Φ-value than another test suite satisfying criterion Z. This would only give an **evidence** that Y yields safer WCET estimates than Z. Consequently, the more samples (i.e., test suites) are compared, the more trust we gain that we can generalize safety assertions for test suites to coverage criteria or generation techniques. Thus, for each coverage criterion or input data generation method under scrutiny, we produce 10 different test suite instances and compute the corresponding expectation value $E(X)$ of a quantity X and the standard deviation $\sigma(X)$. This should minimize the probability that we draw conclusions from an extreme case. Further, a low standard deviation suggests that an input data generation technique is *stable* as the jitter in relative safety is low. Consequently, in order to give a recommendation for using a specific input data generation technique in the context of MBTA, both $E(\Phi(\Gamma))$ and $\sigma(\Phi(\Gamma))$ must be low for many different test suites Γ that are produced by the respective input data generation technique. The same holds for coverage criteria.

As we investigate only one target architecture and a restricted set of benchmarks, we cannot claim that the observations and conclusions we encounter in our experiments are valid for real-time systems in general. Still, we are confident that the TriCore 1796 microprocessor and our benchmarks are representative for the automotive domain.

B. Experimental Results for Coverage Criteria from the Domain of Functional Testing

Regarding the experimental results shown in Table II we can observe the following:

1) There are δ-CFG vertices that are not exercised using random input data generation in all benchmarks except `bsort`. There is no evidence that increasing the amount of random test vectors will guarantee that all vertices are exercised eventually as the experiments show almost no difference between test suits of size 125 and 1000 in benchmarks `bs` and `lift_control`.
2) For benchmark `bsort`, test suites from random input data generation yield much better Φ values than both basic block coverage B and the union of basic block, condition and decision coverage $B \cup C \cup D$. In these examples we use FQL queries of the form

   ```
   IN @FUNC(foo) cover @BASICBLOCKENTRY
   ```

   to request only basic block coverage in a function `foo`. With the query

   ```
   cover (@BASICBLOCKENTRY|@CONDITIONEDGE)
   ```

   we specify the combination of basic block and condition coverage and omit the restriction to function `foo`.

From (1) we can conclude that using random input data generation exclusively in MBTA is problematic as it violates our basic assumption of not modeling system hardware: if parts of the program are not exercised, MBTA lacks data in the timing composition phase. The user would have to insert the data which requires detailed knowledge of the hardware.

Both basic block coverage B and the union of basic block, condition and decision coverage $B \cup C \cup D$ guarantee that each vertex is exercised at least once which is a great advantage over random input data generation. Still, from observation (2) we can conclude that B, $B \cup C \cup D$ as well as MC/DC (MC/DC is subsumed by condition/decision coverage in our experimental setup due to the discussed code modifications) are not always the best choice.

C. Experimental Results for Pair Coverage

We have two motivations now to come up with a coverage criterion that is superior to coverage metrics like B, $B \cup C \cup D$ and MC/DC in the context of MBTA:

1) Betts et al. propose pairwise pipeline coverage in [18] to improve the coverage for pipelines. Their explanatory examples to show the benefits of pairwise pipeline coverage seem promising. However, experiments to back up the claims have not been performed yet.
2) The evaluation of B, $B \cup C \cup D$ and MC/DC show that there is a potential for improvement with respect to the relative safety metric Φ.

We evaluated both basic block pair coverage B^2 which requires that all feasible pairwise combinations of basic blocks are exercised at least once, and $(B \cup C \cup D)^2$, which requires that all feasible pairwise combinations of basic blocks, condition and decision branches are exercised at least once. To specify pair coverage we use FQL queries of the form

```
IN @FUNC(foo) cover @BASICBLOCKENTRY ->
@BASICBLOCKENTRY
```

where "->" states that arbitrary statements may be executed before the second basic block of a pair is reached.

Pairwise pipeline coverage as proposed by Betts et al. is subsumed by both B^2 and $(B \cup C \cup D)^2$. Furthermore, B^2 is obviously subsumed by $(B \cup C \cup D)^2$.

We use the same experimental setup as before and show the results in Table II. The experiments show that both pair coverage criteria B^2 and $(B \cup C \cup D)^2$ outperform their basic counterparts B and $B \cup C \cup D$ considerably for all benchmarks except `bsort` where all criteria perform equally

[4]http://code.forsyte.de/fshell

Benchmark	Test suite Γ_k	$E(\|\Gamma_k\|)$	$E(t_{gen})$ [s]	E(#mv)	$E(\Phi_{\mathcal{P}}(\Gamma_k))$ [ns]	$\sigma(\Phi_{\mathcal{P}}(\Gamma_k))$ [ns]
`bs`	Random input data generation	125	0.4s	2	8039	12.89
$\|V\| = 14$	Random input data generation	250	0.1s	2	8027	15.89
	Random input data generation	500	0.1s	2	8035	10.69
	Random input data generation	1000	0.2s	2	8019	20.16
	B	1.7	0.2s	0	614	182.99
	$B \cup C \cup D$	1.5	0.3s	0	802	134.93
	B^2	2.5	0.6s	0	397	194.15
	$(B \cup C \cup D)^2$	2.5	0.6s	0	395	104.64
	Balanced path generation I	16.1	5.8s	0	122	16.60
	Balanced path generation II	54.9	22.5s	0	87	9.94
`bsort`	Random input data generation	125	0.2s	0	245	106.01
$\|V\| = 15$	Random input data generation	250	0.3s	0	175	102.32
	Random input data generation	500	0.5s	0	108	117.16
	Random input data generation	1000	0.9s	0	100	92.80
	B	1.1	3.9s	0	2337	242.48
	$B \cup C \cup D$	1.0	2.6s	0	2238	191.60
	B^2	1.4	17.0s	0	2307	174.64
	$(B \cup C \cup D)^2$	1.4	18.0s	0	2083	211.25
	Balanced path generation I	12	109.2s	0	1836	16.01
	Balanced path generation II	30.7	99.9s	0	803	662.36
`lift_control`	Random input data generation	125	1.2s	72	240143	1139
$\|V\| = 119$	Random input data generation	250	1.8s	72	240085	1131
	Random input data generation	500	3.4s	72	239501	563
	Random input data generation	1000	6.9s	72	238851	883
	B	37.3	2.2s	0	8812	1474
	$B \cup C \cup D$	37.0	2.1s	0	9560	1495
	B^2	191.8	29.5s	0	4316	888
	$(B \cup C \cup D)^2$	192.2	30.2s	0	4118	893
	Balanced path generation I	220.2	18.1s	0	3656	567
	Balanced path generation II	1107.3	296.6s	0	1879	889
`engine_control`	Random input data generation	125	0.3s	77.6	325540	113851
$\|V\| = 398$	Random input data generation	250	0.6s	56.6	263447	58178
	Random input data generation	500	1.0s	25.1	171492	39299
	Random input data generation	1000	2.0s	20.5	150921	26938
	B	15.6	9.8s	0	100626	1887
	$B \cup C \cup D$	17.5	10.2s	0	100359	2496
	B^2	114.3	191.7s	0	80116	4285
	$(B \cup C \cup D)^2$	112.2	196.6s	0	82496	4154
	Balanced path generation I	178.3	124.9s	0	78950	4590
	Balanced path generation II	482.9	455.0s	0	62709	4226

B: Basic block coverage	C: Condition coverage	D: Decision coverage	$E(X)$: Expectation value for X
$\sigma(X)$: Standard deviation of X	#mv: Number of missed δ-CFG vertices	$\|V\|$: Number of δ-CFG vertices	t_{gen}: test suite generation time

TABLE II
EXPERIMENTAL RESULTS

well. Another benefit of a pair coverage criterion is that the acquired information about **infeasible** pairs can be useful for the *Timing Composition* phase of MBTA. For instance, an IPET problem can be refined in order to exclude those infeasible pairs making the overall analysis more precise.

Still, for both investigated pair coverage criteria, the gap in terms of Φ compared to random input data generation in benchmark `bsort` remains considerable. Consequently, there are WCET conditions for a δ-CFG vertex that cannot be expressed sufficiently by a pair relation. An obvious next step towards a safer coverage metric would be the consideration of triples, quadruples, etc. up to path coverage in the δ-CFG. However, we already run into complexity issues for triple coverage – therefore we have to investigate an alternative approach, which we discuss in the following section.

V. Balanced Path Generation

Recall that without a sophisticated model of the hardware of the target system we cannot derive a perfect coverage metric. All we can do is to react to observations in the experiments and derive criteria and techniques in a best effort manner. Up to now, the experiments show that random input data generation performs best in our setup if the control flow of a program is simple as in `bsort`. However, in general, there are parts of a program that are not exercised if flow of control gets more complex. The idea is now to combine the advantages of random input data generation (efficiency, good values for Φ for those parts of a program that are exercised) with those of $B \cup C \cup D$ or $(B \cup C \cup D)^2$, where all (pairs of) δ-CFG vertices/edges are guaranteed to be exercised at least once. We now introduce two input data generation techniques that both follow this idea: *Balanced Path Generation (BPG) I and II*. Both methods try to efficiently generate many different δ-CFG paths on the one hand and try to exercise all program parts at a **balanced** level on the other hand: not only should there be no δ-CFG vertices that are exercised more frequently than others. Also, for any two measurements of a specific δ-CFG vertex, the history (i.e., the execution sub-paths that end at the δ-CFG vertex) should be different.

A. Balanced Path Generation I

The key idea is to split up the problem of finding a test suite that satisfies $(B \cup C \cup D)^2$ into n subproblems that are processed by FShell consecutively: the first problem (denoted by an FQL expression) describes a subset P_1 of all pairs $P = (P_1 \cup \ldots \cup P_n)$ that are to be covered, the second problem describes subset $P_2 \subseteq P$ that is disjoint from P_1, and so forth. In order to achieve this, we use FQL queries of the form

```
 IN @FUNC(foo) cover (@CONDITIONEDGE |
@BASICBLOCKENTRY) -> (@LINE(45) | @LINE(39)
| @LINE(84))
```

to pair each basic block or condition outcome with lines 45, 39 and 84 only. We then repeat this process with analogous queries for other lines.

After we have input all corresponding n FQL expressions to FShell (i.e., we run FShell n times) the union of all test suites satisfies $(B \cup C \cup D)^2$ on the one hand. More importantly, by partitioning the problem we end up with more generated test vectors because FShell is prevented from minimizing the size of the resulting test suite (by partitioning the test suite specification FShell lacks context information for checking which test goals are satisfied by a test vector candidate). The experiments in Table II show that these additional test vectors induce a positive effect in terms of safety: for all benchmarks BPG I is rated to give safer local WCET estimates than B^2 and $(B \cup C \cup D)^2$ in our experimental setup.

B. Balanced Path Generation II

Generating test suites that satisfy pair coverage has one drawback: many combinations of, e.g., basic blocks are infeasible. Although this information can be quite useful for the Timing Composition phase of MBTA, it prevents us from generating many test vectors efficiently. Proving infeasibility by a model checker is usually more computationally expensive than finding a test vector for a feasible property. Consequently, FShell spends most of its time proving infeasibility (yielding no test vectors) when generating a test suite that satisfies pair coverage.

Now, the idea is to generate a test suite that satisfies $B \cup C \cup D$ for which most of the test goals FShell has to process are feasible. This implies that the number of generated test vectors per execution unit of FShell is high. With the objective to maximize the amount of test vectors, BPG II generates 30 **different** test suites (FShell offers an option to yield mutually different test suites for an FQL expression), each satisfying $B \cup C \cup D$. Finally, the union of all 30 test suites forms the result (our experiments revealed that 30 is an effective number for the kind of benchmarks we investigated).

The experiments illustrate that Balanced Path Generation II provides the best results in terms of Φ except for benchmark `bsort` where random input data generation remains unbeaten. The seemingly large expectation value of 62709 nanoseconds for Φ in benchmark `engine_control` is still admissable as the program is the largest in our experiment with 398 δ-CFG vertices. The average observed underestimation per node is therefore only $\frac{62709}{398 \times 1000} = 0.16$ microseconds.

VI. Summary, Conclusion and Future Work

We illustrated the dilemma of investigating the safety of measurement-based timing analysis (MBTA) without spending the arduous effort for studying and modeling the system hardware.

Using a relative safety metric and a set of representative benchmarks, we have shown empirically for the TriCore 1796, a commonly used microprocessor in the automotive domain, that structural code-coverage criteria from the domain of functional testing like basic-block coverage, condition coverage, decision coverage, or modified decision-decision coverage (MC/DC) are not suited for measurement-based timing analysis if safety of WCET estimates is of uttermost concern.

Further we examined coverage criteria that require all feasible pairs of program entities to be exercised in combination. They are shown to be superior to code-coverage criteria from the domain of functional testing but our experimental setup also reveals that there is still a chance that unsafe WCET estimates are derived.

Finally, we introduced *Balanced Path Generation I and II* and showed by experiments that both perform better in terms of relative safety than all other coverage criteria under investigation.

However, the experiments also demonstrated that by using *Balanced Path Generation* there is still a chance that local WCET estimations are unsafe. Its improvement is therefore subject to further research. One idea is to combine model checking with heuristic search methods like genetic algorithms.

Another open issue is that we still lack a coverage metric for MBTA. We think that because such a coverage metric is very hard (if not impossible) to derive without using a sophisticated hardware model, it might be more productive to define a convergence criterion for input data generation techniques. The definition, motivation and evaluation of such a criterion is also part of future work.

Acknowledgment

The research leading to these results has received funding from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13. We also thank Andreas Holzer for the C Modeler to obtain δ-CFGs, Mohammad Khaleghi for the random input data generator and Benedikt Huber for the inspiring and constructive discussions.

REFERENCES

[1] A. Burns and A. J. Wellings, *Real-Time Systems and Programming Languages*, 2nd ed. Addison Wesley, 1996.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[3] R. Kirner, I. Wenzel, B. Rieder, and P. Puschner, *Intelligent Systems at the Service of Mankind*. Augsburg, Germany: UBooks Verlag, Jan. 2006, vol. 2, ch. Using Measurements as a Complement to Static Worst-Case Execution Time Analysis, pp. 205–226, iSBN: 3-86608-052-2.

[4] S. Stattelmann and F. Martin, "On the use of context information for precise measurement-based execution-time estimation," in *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, B. Lisper, Ed. Austrian Computer Society, July 2010, pp. 68–79.

[5] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, Oct. 2008.

[6] M. Zolda, S. Bünte, and R. Kirner, "Towards adaptable control flow segmentation for measurement-based execution time analysis," in *Proc. 17th International Conference on Real-Time and Network Systems (RTNS)*, Paris, France, Oct. 2009.

[7] A. Betts and G. Bernat, "Tree-based WCET analysis on instrumentation point graphs," in *Proc. 9th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Gyeongju, Korea, Apr. 2006.

[8] J. Wegener and M. Grochtmann, "Verifying timing constraints of real-time systems by means of evolutionary testing," *Real-Time Systems*, vol. 15, no. 3, pp. 275–298, 1998.

[9] J. Wegener and F. Mueller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, November 2001. [Online]. Available: http://www.springerlink.com/content/l64485g1774x7674/

[10] P. Puschner, "Worst-case execution time analysis at low cost," in *Proc. 14th IFAC Workshop on Distributed Computer Control Systems*, Jul. 1997, pp. 16–21.

[11] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *DAC '95: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1995, pp. 456–461.

[12] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[13] M. Zolda, S. Bünte, and R. Kirner, "Context-sensitivity in IPET for measurement-based timing analysis," in *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'10)*, Oct. 2010.

[14] S. Ntafos, "A comparison of some structural testing strategies," *IEEE Transactions on Software Engineering*, vol. 14, pp. 868–874, 1988.

[15] M. Roper, *Software Testing*. New York, NY, USA: McGraw-Hill, Inc., 1995.

[16] I. RTCA, "DO-178B: Software considerations in airborne systems and equipment certification," *Requirements and Technical Concepts for Aviation*, 1992.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 2002, pp. 58–70.

[18] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "WCET coverage for pipelines," RealTime Systems Research Group - University of York and Institute of Computer Engineering - Vienna University of Technology, Tech. Rep., 2006.

[19] A. Colin and S. Petters, "Experimental evaluation of code properties for wcet analysis," dec. 2003, pp. 190 – 199.

[20] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a Java processor," *Software: Practice and Experience*, vol. 40/6, pp. 507–542, 2010.

[21] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "How did you specify your test suite ?" in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, Sep. 2010.

[22] ——, "Fshell: Systematic test case generation for dynamic analysis and measurement," in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008)*, ser. Lecture Notes in Computer Science, vol. 5123. Princeton, NJ, USA: Springer, July 2008, pp. 209–213.

[23] ——, "Query-driven program testing," in *Proceedings of the Tenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, ser. Lecture Notes in Computer Science, N. D. Jones and M. Müller-Olm, Eds., vol. 5403. Savannah, GA, USA: Springer, January 2009, pp. 151–166.

[24] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.