# Context-Sensitive Measurement-Based Worst-Case Execution Time Estimation

Michael Zolda, Sven Bünte
*Department of Computer Engineering*
*Vienna University of Technology*
*Vienna, Austria*
*Email: {michaelz, sven}@vmars.tuwien.ac.at*

Raimund Kirner
*Department of Computer Science*
*University of Hertfordshire*
*Hatfield, United Kingdom*
*Email: r.kirner@herts.ac.uk*

***Abstract*—The goal of measurement-based WCET estimation (MBWE) is to derive an estimate of the worst-case execution time (WCET) of a given piece of software on a particular target platform by executing the software on the target hardware and analyzing the obtained time-stamped execution traces.**

**In this paper we introduce context-sensitive MBWE, an approach that can reduce pessimism by making use of state information that is exposed through individual control-flow decisions. We show how to extend the popular IPET method, to obtain tighter WCET estimates. We provide confirmative empirical results that demonstrate the effectiveness of our approach.**

***Keywords*-WCET analysis, worst-case execution time, IPET, measurement-based timing analysis, temporal analysis, non-functional testing**

## I. Introduction

In *Measurement-based WCET estimation* (MBWE), which is a special case of *measurement-based timing analysis* (MBTA) [1], we estimate the WCET of a given piece of software by executing it with selected input vectors on the actual target hardware and processing the obtained *time-stamped execution traces* to infer a WCET estimate.

In traditional MBWE we use the *maximal observed execution time* (MOET) of individual program parts as an estimate for the local WCET. The accuracy of the overall WCET estimate is influenced by two opposed effects:

*Optimism* arises due to the coverage limit of the timing-relevant computer state (TRCS) [2]—the relevant part of the system state that can influence the execution time of an individual instruction or instruction sequence—that we face during measurement. It is an inherent property of the measurement-based approach that we can never guarantee that the local worst-case behavior of individual program parts is always covered by a time-stamped trace.

*Pessimism* arises due to abstraction during temporal analysis. Traditional IPET [3], [4] considers only the highest of all possible execution times of each CFG node. However, due to functional dependencies in the program as well as in the hardware, any local WCET scenario need not necessarily contribute to the global WCET [2].

In practice, we can observe that these two effects typically tend to compensate each other to a certain degree. However, in the experiments that we performed we saw that pessimism virtually always appears to overcompensate optimism.

Unlike in static WCET analysis, estimates obtained by MBWE are not directly useable for verification or certification. The most important use-case of MBWE 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 codes 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, even if static WCET analysis is not viable, e.g., due to high initial costs to develop a dependable analysis for a specific target platform, no or limited retargetability, or highly pessimistic WCET bounds.

In this paper we attack the problem of pessimism. As our first contribution, we present strong evidence that the control flow decisions that are taken just before some program part is executed can expose a substantial part of the TRCS. Based on this evidence, we develop, as our second contribution, the idea of using the information that is exposed through individual control-flow decisions to reduce pessimism. We show how the context-sensitive approach can be integrated in IPET and conclude with confirmative experimental results.

## II. Preliminaries

In MBWE we try to derive a WCET estimate for transformational tasks, i.e., tasks that initially read their complete input, execute for a finite amount of time, and finally produce some output before terminating. Such tasks are the usual building blocks of more complex systems [5]. Following the state-of-the-art in WCET estimation, we assume that the software under scrutiny is running on a single CPU core without interference from other tasks or hardware features like dynamic frequency scaling.

We analyze at the source code level, where we can benefit from features like type system and high-level control flow structures. However, this decision makes it necessary to consider code transformations that happen at compile time. Recent studies of compiler transformations show that only certain transformations are problematic with respect to MBWE [6], [7]. These can be deactivated without significant

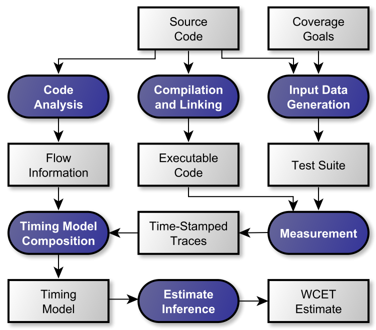

Figure 1. The workflow of measurement-based WCET estimation.

performance loss. Another option is to apply *flow transformations* to simulate the compiler transformations [8].

### A. The MBWE Workflow

We briefly describe the individual steps of the MBWE workflow, as indicated in Figure 1:

*Code Analysis* takes the source code of the software under scrutiny and applies static program analysis techniques to extract the control flow graph (CFG). Code analysis can also derive additional flow information, like iteration/recursion bounds for looping/recursive control flow.

*Compilation and Linking* takes the source code and produces executable code for the target platform.

*Input Data Generation* produces a suitable set of input vectors—a *test suite*—to be used in the subsequent measurement step. *Suitable* means that the test suite should conform to a given *coverage goal*. That can be a structural coverage goal, e.g., a requirement like basic block, condition or decision coverage, or more sophisticated specifications [9], [10]. Alternatively, a coverage goal can be an optimization goals, e.g., maximization of locally observed execution times [11].

*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. Possible measurement techniques comprise non-intrusive capturing of traces via special debugging interfaces [12], [13], [14], software instrumentation, and cycle-accurate simulation.

*Timing Model 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.

*Estimate Inference* produces a WCET estimate from the timing model.

### B. Methods for Input Data Generation

It is a basic assumption of MBWE that we do not know the hardware in full detail. Creating a sufficiently detailed model of present-day hardware is an expensive and error-prone task. Specifications that are published by hardware vendors do not normally contain enough detail to model temporal aspects. It can even happen that the hardware vendor is unwilling or unable to provide the required information. MBWE shuns these problems by relying time-stamped execution traces obtained from the actual target system instead of using a complete hardware model.

Because the space of possible input vectors is usually huge, it is not tractable to expose all possible software behavior. A selection has to be made. We use a mix of heuristic techniques that try to minimize the chance of missing relevant temporal behavior [10]:

*Random input data generation* can sometimes achieve a surprisingly good coverage in a very short time, but tends to miss rare conditions.

*Model checking* is good at dealing with special cases, but expensive in terms of time and memory complexity [15].

*Heuristic optimization methods* can do a good job at covering certain rare conditions, without the high time and memory costs of model checking. They cannot guarantee to cover all special cases, though. Finding good heuristic methods is part of an ongoing research effort.

### C. Temporal Analysis using IPET

The analysis we discuss in this paper is based on the *Implicit Path Enumeration Technique* (IPET) with linear programming [3], [4], which views the problem of finding the WCET of a piece of software as an optimization problem: The possible control flow within the software is approximated by linear constraints. The operations performed within the software have different costs in execution time. The optimization goal is to find the maximal overall costs realized by some feasible flow.

First the software is decomposed into atomic parts that can be assigned individual local WCET values. These atomic parts must match the nodes in the flow graph obtained from program analysis. They can be individual machine instructions, basic blocks, or larger software components.

We use *node variables* $f_v$ to capture the control flow through (i.e. the number of executions of) each node $v$ during a single run of the software. Likewise, we use *edge variables* $f_e$ to capture the number of times that control passes through each edge $e$. We link node and edge variables via linear constraints, according to the CFG semantics:

$$f_v = \sum_{f \in out(v)} f_e,$$

i.e., the flow through a node $v$ equals the sum of the flow through its outgoing edges $out(v)$. A similar equation connects each node with its incoming edges.

As the CFG semantics allows for unbounded flow within loops, the number of iterations must be limited. For *natural*

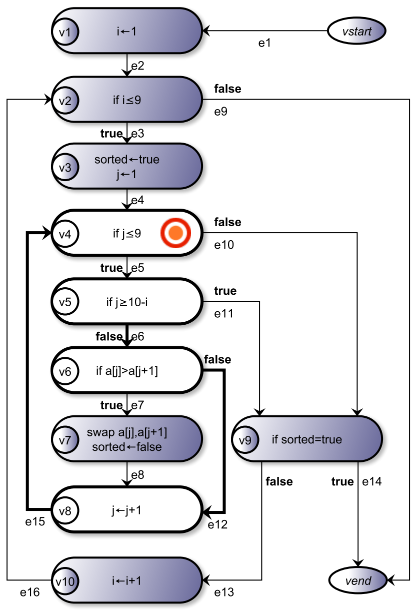

Figure 2. Example CFG of Bubble Sort. Also indicated is a segment comprising nodes $v_4$, $v_5$, $v_6$, $v_8$, and edges $e_6$, $e_{12}$, $e_{15}$, for the marked target node $v_4$, c.f. Section IV.

*loops* these are usually inequalities that bound the flow through the back edge *back* by a multiple $b$ of the total outside flow $f_1, \ldots, f_n$ into the loop header, i.e.,

$$f_{back} \leq b \cdot (f_1 + \ldots + f_n).$$

Many iteration constraints can be derived automatically by code analysis. Otherwise they must be supplied by an expert who has to manually inspect the source code.

The linear problem's objective function is

$$\sum_{v \in nodes} c_v \cdot f_v,$$

where $c_v$ is the execution time cost of node $f_v$. Maximizing this function yields the WCET estimate.

## III. A Motivating Example

Pessimism arises during analysis whenever there are two or more ways that the system can behave, and we abstract from this by assuming the highest execution time. The execution time of each CFG node can vary with the TRCS. In its original form IPET does not consider the TRCS, but rather assumes a constant execution time for each node. If we want to reduce the pessimism of IPET, we have to extend the method so it can distinguish between different TRCS.

Consider Figure 2, depicting the CFG of a *Bubble Sort* implementation. After systematic input data generation and measurement, we inspected the obtained time-stamped execution traces and made the following observations:

1) The execution time of $v_4$ never exceeded 634ns.
2) Whenever edge $e_5$ had been taken, the execution time of the closest subsequent occurrence of $v_4$ did not exceed 440ns.
3) Whenever edge $e_7$ had been taken, the execution time of the closest subsequent occurrence of $v_4$ did not exceed 400ns.
4) Whenever edge $e_{13}$ had been taken, the execution time of the closest subsequent occurrence of $v_4$ did not exceed 379ns.

We have observed similar patterns in various experiments with different benchmarks. We therefore conclude that the control flow leading to the execution of a node $v$ can potentially expose a substantial part of the TRCS for $v$—enough information to considerably tighten the MOET of the target node $v$. By adapting IPET to handle control flow individually, we can effectively reduce pessimism.

On the downside, the segregation of a special case can lead to a local increase of optimism. This can happened when some identified special case has not received sufficient coverage. For instance, in the previous example we observed that whenever edge $e_{13}$ had been taken in an execution path, the execution time of the closest subsequent occurrence of $v_4$ did not exceed 379ns. However, there might exist some runs through $e_{13}$ that induce a considerably higher execution time in the closest subsequent occurrence of $v_4$ which just have not been covered during input data generation and measurement. In that case singling out the observed special behavior would introduce undue local optimism.

Our current implementation uses model-checking based input data generation to increase the coverage for each identified special case. Thus we either increase the confidence in the special case, or refute it by a higher execution time.

Traditional IPET presumes constant costs for each node, namely the absolute worst case costs. In the measurement-based approach we use the local MOET as an estimate of the local WCET. In our example we would therefore obtain an objective function of the form

$$\ldots + c_{v_3} \cdot f_{v_3} + 614 \cdot f_{v_4} + c_{v_5} \cdot f_{v_5} + \ldots$$

In our extended version of IPET we introduce new *flow variables* $f^{v_4}_{634}, f^{v_4}_{440}, f^{v_4}_{400}, f^{v_4}_{379}$ that describe the individual flows that induce different MOETs in $v_4$. Each execution of $v_4$ is associated with exactly one of the control flows:

$$f^{v_4}_{634} + f^{v_4}_{440} + f^{v_4}_{400} + f^{v_4}_{379} = f_{v_4}.$$

In the original objective function we replace the term $614 \cdot f_{v_4}$ with the sum

$$614 \cdot f^{v_4}_{634} + 440 \cdot f^{v_4}_{440} + 400 \cdot f^{v_4}_{400} + 379 \cdot f^{v_4}_{379}$$

As the program can never execute backwards, there should, of course, be no negative flows, so we add appropriate non-negativity constraints

$$f^{v_4}_t \geq 0, \quad \text{for } t \in \{634, 440, 400, 379\}.$$

At this point our modified IPET problem yields the same result as the original problem, as the maximization of the objective function will attribute all executions of $v_4$ to the costliest flow, i.e., $f_{440}^{v_4} = f_{400}^{v_4} = f_{379}^{v_4} = 0$ and $f_{634}^{v_4} = f_{v_4}$.

Next we specify upper and lower bounds for the individual flows. We translate our observations individually, working upwards from the lower execution times bounds:

1) Variable $f_{379}^{v_4}$ should describe the flow from edge $e_{13}$ to node $v_4^{v_4}$ that is not shared with any cheaper flow. Since $f_{379}^{v_4}$ is the cheapest flow, that would be the flow through edge $e_{13}$ that does not escape through edge $e_9$. That flow can be bound through the constraints

$$f_{379}^{v_4} \geq f_{e_{13}} - f_{e_9}, \quad f_{379}^{v_4} \leq f_{e_{13}} + f_{e_2} - f_{e_9},$$
$$f_{379}^{v_4} \leq f_{e_{13}}.$$

2) Variable $f_{400}^{v_4}$ should describe the flow from edge $e_7$ to node $v_4$ that is not shared with any cheaper flow. Because $v_4$ post-dominates $e_7$ and flow $f_{400}^{v_4}$ is disjoint from the only cheaper flow $f_{379}$, that would be the complete flow through edge $e_7$, i.e.,

$$f_{400}^{v_4} = f_{e_7}.$$

3) Variable $f_{440}^{v_4}$ should describe the direct flow from edge $e_5$ to node $v_4$ that is not shared with any cheaper flow. That would be the flow through edge $e_5$ that does neither escape through edge $e_9$ or $e_{14}$, nor is shared with $f_{400}^{v_4}$ or $f_{379}^{v_4}$:

$$f_{440}^{v_4} \geq f_{e_5} - f_{e_{14}} - f_{e_7} - f_{e_{13}}, \quad f_{440}^{v_4} \leq f_{e_5}.$$

4) Variable $f_{634}^{v_4}$ should describe the direct flow from edge $e_1$ to node $v_4$ that cannot be described by any of the cheaper flows. Because the flow $f_{634}^{v_4}$ is disjoint from all cheaper flows, that would be the flow through edge $e_1$ that does not escape through edge $e_9$:

$$f_{634}^{v_4} \geq f_{e_1} - f_{e_9}, \quad f_{634}^{v_4} \leq f_{e_1} + f_{e_{16}} - f_{e_9},$$
$$f_{634}^{v_4} \leq f_{e_1}.$$

With these constraints in place the modified IPET problem for our example yields a less pessimistic WCET estimate that takes into account variations in the execution time of node $v_4$. We could proceed likewise for the other CFG nodes.

## IV. Context-Sensitive MBWE

Having presented the main idea of our approach on a motivating example, we now present the systematic details.

Let $G = \langle V_G \uplus \{v_{start}, v_{end}\}, E_G \rangle$ be a CFG, where $V_G \uplus \{v_{start}, v_{end}\}$ is the set of *nodes*, including a virtual *start node*, $v_{start}$, and a virtual *end node*, $v_{end}$, and where $E_G \subseteq (V_G \uplus \{v_{start}\}) \times (V_G \uplus \{v_{end}\})$ is the set of directed *edges*.

A *straight path $\pi$ from a node $w$ to a node $v$* is a path $v_1 \ldots v_n$ through $G$ with $v_1 = w$, $v_n = v$, where $v_i \notin \{v, w\}$, for $1 < i < n$.

A *straight path $\pi$ from an edge $e$ to a node $v$* is a path $v_1 \ldots v_n$ through $G$, with $(v_1, v_2) = e$, $v_n = v$, where $(v_i, v_{i+1}) \neq e$, for $1 < i < n-1$, and where $v_i \neq v$, for $1 < i < n$.

A *history-sensitive execution time of a node $v$* is a pair $(\pi, t)$, where $\pi$ is a straight path to $v$, called *history path*, and where $t$ is an execution time of $v$.

For context-sensitive MBWE, we start by generating test data, performing measurements, building an initial database of history-sensitive execution times. Next, we classify the different executions times of a given node based on control flow decisions in its execution history. It is therefore important that the database can provide a suitable collection of different history paths for each CFG node. Hence, we impose the following coverage criterion:

> For each node $v$ and edge each $e$, we demand at least one history-sensitive execution time $(\pi, t)$ where $\pi$ is a straight path from $e$ to $v$.

This is a minimal requirement. Ideally, we should obtain a large number of history-sensitive execution times for each edge-node pair.

Let $M$ be a database of observed context-sensitive execution times for a node $v$, and let $x$ be either a node or an edge. The *$x$-dominated maximal observed execution time $MOET(x, v)$ of $v$* (where $x$ is either a node or an edge) is the maximal execution time observed in $v$ over all observed history-sensitive execution times $(\pi, t) \in M$ where $\pi$ is a straight path from $x$ to $v$, i.e.,

$$MOET(x, v) = \max\{t \mid (\pi, t) \in M, \pi \text{ is a straight path from } x \text{ to } v\}.$$

The central assumption of our approach is that $MOET(x, v)$ is a suitable estimate for all possible history-sensitive execution times $(\pi, t)$ of $v$, where $\pi$ is any straight path from $x$ to $v$. As mentioned before, this is an optimistic assumption that needs support through appropriate coverage.

How can context-sensitive execution times be used in IPET? To answer this question, we first introduce the general concept of a segment to describe sets of straight paths. We then show how the flow through a segment, i.e., the flow through the corresponding set of paths, can be approximated by a set of IPET constraints. Next, we explain how to construct a segment that describes the set of straight paths from an edge $e$ to a node $v$. Subsequently, we show how to subtract, from a given segment $S$ that describes the set of straight paths from an edge $e$ to a node $v$, another segment $T$ that describes the set of straight paths from an edge $e'$ to $v$, where $e'$ is inside segment $S$. This operation is sufficient to exclude cheaper subsets of straight paths to $v$, as was seen in our motivating example from Section III. We complete our discussion of context-sensitive IPET with a sketch of an algorithm that can identify segment nestings that correspond to increasingly cheaper subsets of straight paths.

A segment is a weakly connected subgraph of the CFG with a defined set of entry edges that point into the subgraph and a defined set of exit edges that point out of the subgraph. Its purpose is to capture the total flow through the subgraph from any of its entry edges to any of its exit edges. Any flow coming into the subgraph though an edge that is not an entry edge, as well as any flow going out of the subgraph through an edge that is not an exit edge does *not* contribute to the segment flow.

A *segment* is a quadruple $\langle V, E, \mathit{Entry}, \mathit{Exit}\rangle$ comprising a set of nodes $V \subseteq V_G$, a set of edges $E \subseteq E_G$, such that $S = \langle V, E\rangle$ is a weakly connected graph, a set of *entry edges* $\mathit{Entry} = \{(v, w) \mid (v, w) \in (E_G \setminus E), w \in V\}$, and a set of *exit edges* $\mathit{Exit} = \{(v, w) \mid (v, w) \in (E_G \setminus E), v \in V\}$.

As a shorthand notation, let $\overline{\mathit{Entry}}$ be the set of CFG edges that point into the segment from outside, and let $\overline{\mathit{Exit}}$ be the set of CFG edges that point out from inside of the segment, but are no exits, i.e.,

$$\overline{\mathit{Entry}} = \{(v, w) \mid (v, w) \in (E_G \setminus E) \setminus \mathit{Entry}, w \in V\},$$
$$\overline{\mathit{Exit}} = \{(v, w) \mid (v, w) \in (E_G \setminus E) \setminus \mathit{Exit}, v \in V\}.$$

For example, in Figure 2 we have highlighted segment

$$P = \langle V_P, E_P, \mathit{Entry}_P, \mathit{Exit}_P\rangle, \text{ where}$$
$$V_P = \{v_4, v_5, v_6, v_8\}, \quad E_P = \{e_6, e_{12}, e_{15}\},$$
$$\mathit{Entry}_P = \{e_5\}, \quad \mathit{Exit}_P = \{e_5, e_{10}\},$$
$$\overline{\mathit{Entry}_P} = \{e_4, e_8\}, \quad \overline{\mathit{Exit}_P} = \{e_7, e_{11}\}.$$

The flow $f$ through a segment $\langle V, E, \mathit{Entry}, \mathit{Exit}\rangle$ can be approximated by the following constraints:

$$f \le \sum_{x \in \mathit{Entry}} f_x + \sum_{y \in \overline{\mathit{Entry}}} f_y - \sum_{z \in \overline{\mathit{Exit}}} f_z,$$
$$f \le \sum_{x \in \mathit{Entry}} f_x, \qquad f \ge \sum_{x \in \mathit{Entry}} f_x - \sum_{y \in \overline{\mathit{Exit}}} f_y.$$

For instance, the flow through our example segment $P$ can be approximated by the constraints

$$f_S \le f_{e_5} + f_{e_4} + f_{e_8} - f_{e_7} - f_{e_{11}},$$
$$f_S \le f_{e_5}, \quad f_S \ge f_{e_5} - f_{e_7} - f_{e_{11}}.$$

To construct a segment

$$S(e, v) = \langle V(e, v), E(e, v), \mathit{Entry}(e, v), \mathit{Exit}(e, v)\rangle,$$

that describes the straight paths from an edge $e$ to a node $v$, let $V(e, v)$ and $E(e, v)$, respectively, be the sets of all nodes and edges reachable from $e$ without passing through $e$ or $v$, and from where $v$ can be reached, let $\mathit{Entry}(e, v) = \{e\}$, and let $\mathit{Exit}(e, v)$ be the set of all outgoing edges of $v$.

The flow through this segment corresponds to the number of history-sensitive executions of $v$ with a straight history path $\pi$ from $e$ to $v$, i.e., those executions of $v$ for which we have assumed the $e$-dominated maximal observed execution time $\mathit{MOET}(e, v)$. For instance, Segment

$$R = \langle V_R, E_R, \mathit{Entry}_R, \mathit{Exit}_R\rangle, \text{ where}$$
$$V_R = \{v_4, v_7, v_8\}, \quad E_R = \{e_8, e_{15}\},$$
$$\mathit{Entry}_R = \{e_7\}, \quad \mathit{Exit}_R = \{e_5, e_{10}\},$$
$$\overline{\mathit{Entry}_R} = \{e_4, e_{12}\}, \quad \overline{\mathit{Exit}_R} = \emptyset.$$

describes the straight paths from edge $e_7$ to node $v_4$, i.e., it is the segment for the flow $f_{400}^{v_4}$ from motivating example in Section III.

To construct from two given segments

$$S = \langle V_S, E_S, \mathit{Entry}_S, \{\mathit{out}(v)\}\rangle \text{ and}$$
$$T = \langle V_T, E_T, \mathit{Entry}_T, \{\mathit{out}(v)\}\rangle,$$

where $\mathit{out}(v)$ is the set of outgoing edges of node $v$, $V_T \subseteq V_S$, $E_T \subseteq E_S$, $\mathit{Entry}_T \subseteq E_S$, a segment

$$U = \langle V_U, E_U, \mathit{Entry}_U, \mathit{Exit}_U\rangle$$

that captures the flow through $S$ that does not also contribute to the flow through $T$, let $V_U$ and $E_U$, respectively, be the sets of all nodes and edges that are reachable from any edge in $\mathit{Entry}_S$ without passing through any edge in $\mathit{Entry}_T$ or through node $\{v\}$, and from where $v$ can be reached without passing through any edge in $\mathit{Entry}_T$, let $\mathit{Entry}_U = \mathit{Entry}_S$, and let $\mathit{Exit}_U = \{\mathit{out}(v)\}$.

This construction rule allows us to exclude nested flows from surrounding flows, like in our motivating example.

There remains one important open point: How can we find a segmentation for each CFG node that can effectively reduce pessimism in our analysis? We sketch an algorithm that produces a nested segmentation for any given target node $v$ that can be resolved by nesting rule we just introduced, with separate segments starting in all edges $e = (u, w)$ with an $e$-dominated maximal observed execution time $\mathit{MOET}(e, v)$ of $v$ that is lower than the $u$-dominated maximal observed execution time $\mathit{MOET}(u, v)$ of $v$.

Our algorithm performs a depth first search along the transpose graph $G'$ of $G$, starting from the target node $v$, i.e., the algorithm searches backwards from $v$ until it runs into an edge $e = (u, w)$ that is either a start edge ($u = v_{start}$), a back edge, a forward edge, or a cross edge. In either of these cases it creates a new segment entry at edge $e$, for a segment with associated costs of $\mathit{MOET}(e, v)$. Upon backtracking the algorithm collects the nodes $u$ and edges $e = (u, w)$ along the backtracking path into all segments for which entries have been created in the currently finished search subtree, unless it finds $\mathit{MOET}(e, v) < \mathit{MOET}(u, v)$. In that case it finishes all those segments at edge $e$ ($u$ is the last node that is collected into the segments) and starts a new segment entry at edge $e$, for a segment with associated costs of $\mathit{MOET}(e, v)$.

Table I
WCET ESTIMATES AND END-TO-END MAXIMUM OBSERVED EXECUTION TIME FOR EACH BENCHMARK.

| Benchmark | | WCET Estimate | | |
|---|---|---|---|---|
| Suite | Function | Trad. | Sens. | MOET |
| MD | binary_search | 25.7$\mu$s | 20.7$\mu$s | 13.0$\mu$s |
| MD | bsort10 | 333.2$\mu$s | 222.5$\mu$s | 174.4$\mu$s |
| PB-F1a | servo_set | 230.0$\mu$s | 228.4$\mu$s | 200.3$\mu$s |
| PB-F2 | vector_10 | 9.6$\mu$s | 9.3$\mu$s | 9.1$\mu$s |
| JOP-Lift | ctrl_loop | 57.6$\mu$s | 45.5$\mu$s | 43.0$\mu$s |

Table II
CODE METRICS AND TIME SPENT ON GENERATING INPUT DATA.

| Benchmark | | CFG Size | | Input Data |
|---|---|---|---|---|
| Suite | Function | Nodes | Edges | Generation |
| MD | binary_search | 14 | 16 | 37s |
| MD | bsort10 | 15 | 19 | 7134s |
| PB-F1a | servo_set | 43 | 56 | 222s |
| PB-F2 | vector_10 | 18 | 21 | 5s |
| JOP-Lift | ctrl_loop | 119 | 175 | 81270s |

After the depth first search has completed, all bogus segment entries, i.e., entry edges $e = (u, w)$ with $MOET(e, v) = MOET(u, w)$ are removed and turned into inner edges of the corresponding segment.

For each segment $S$ that was created by the algorithm, we let $Exit_S$ be the set of all outgoing edges of $v$.

## V. EVALUATION

We have implemented both, traditional and context-sensitive IPET within the FORTAS timing analysis suite [16]. To evaluate the benefit that context-sensitive IPET can provide, we have performed a WCET estimation on five benchmark programs taken from different sources.

Table I shows our WCET estimates and end-to-end MOET. The latter is our best lower bound of the actual WCET.

Table II views some code metrics, as well as the analysis time spent on generating input data. Due to the concurrent, multi-user, server-client architecture of our timing analysis framework, it is difficult to provide precise numbers for the total analysis times. However, as input data generation is by far the costliest step in our analysis, the time spent on this part provides a rough estimate of the total analysis effort.

All analyses were performed on an Intel Core2 Quad Q9450 CPU running at 2.66GHz with 8GiB of DRAM. The analyzed code was taken from the *Mälardalen WCET Benchmark Suite* (MD) [17], *PapaBench* (PB) [18], as used in the *WCET Tool Challenge 2011*[1], and the *Java Optimized Processor* Benchmark Suite (JOP)[2].

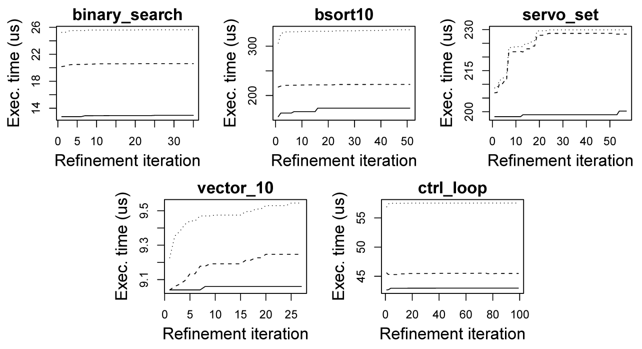

Figure 3. Convergence of WCET estimates obtained with traditional IPET (dotted line), context-sensitive IPET (dashed line), and end-to-end MOET (solid line) during refinement.

Our target platform was a TriBoard TC1796 evaluation board[19] equipped with a TriCore 1796 processor [20]. We used a *Lauterbach PowerTrace* [12] device to capture complete end-to-end time-stamped execution traces of the software via the Infineon OCDS interface, without exerting a probe effect. The hardware setup was the same as the one used during the WCET Tool Challenge [21].

To generate appropriate input data, we used *FShell*, a *CBMC*-based [22] model checker that generates test suites from formal test suite specifications [9], [23].

For each benchmark, we initially generated a test suite matching the coverage criterion introduced in Section III, i.e., a test suite that contains a test case for each feasible edge-node pair. We then obtained the corresponding traces by measurement and constructed a segmentation. We then started to refine our timing model by generating additional test data for each segment. In the refinement phase we generated additional test suites for each segment, containing at least one test case for each feasible edge-edge-node triple on any path through the segment.

The results in Table I show the final WCET estimates after the last refinement step. However, users of our analysis framework need not wait for the whole analysis process to complete. Rather, they can obtain an intermediate WCET estimate at any time during refinement. As can be seen in Figure 3, good estimates can usually be obtained after just a few refinement steps.

## VI. RELATED WORK

Theiling and Ferdinand use abstract interpretation to classify instruction cache accesses into the categories *always hit*, *always miss*, *persistent*, and *not classified* [24]. By using *virtual inlining/virtual unrolling* (VIVO), they are able to separate the first iteration of loops from subsequent ones,

[1]The benchmarks for the WCET Tool Challenge can be found at http://www.mrtc.mdh.se/projects/WCC/2011.

[2]Function *ctrl_loop* is taken from a C port of the *lift control* application from the JOP Benchmark Suite. The original Java source is available at http://www.soc.tuwien.ac.at/trac/jop/browser/java/target/src/bench/jbe/lift.

and to distinguish procedure invocations that originate from different call sites. They perform IPET on a corresponding partially unrolled and inlined CFG. The different copies of control flow nodes can then be weighted according to their context-sensitive cache classification.

The concept of VIVO is not limited to the static analysis approach of timing analysis, but can also be applied in the measurement-based approach. Through partial loop peeling we can obtain a control flow graph that exposes certain loop iterations—usually the first one. It is also easy to conceive a generalized variant of VIVO that can expose subgraphs or even individual paths. We presented such an approach in [16]. However, this strategy is limited by a potentially exponential growth of the flow graph.

The approach that we present in this paper does not increase this size of the CFG at all. We merely modify the IPET problem based on observed correlations. In a practical analysis tool, using a hybrid approach that combines the benefit of both approaches might be a good choice.

Betts and Bernat present a tree-based calculation method for measurement-based timing analysis that combines execution time measurements of complete basic block sequences instead of treating them individually [25]. By treating certain sequences of basic blocks as atomic, the method is able to benefit from a certain reduction of pessimism. In a similar fashion, Wenzel et al. treat entire subgraphs as atomic units [1]. Our approach attacks the problem of pessimism in an entirely different fashion: We do not rely on the combined execution time of multiple CFG nodes to achieve a reduction of pessimism. Rather, we use information from the execution history to discriminate different execution times of individual CFG nodes. More recent work of Betts et al. on measurement-based timing analysis focuses on the integration of object-code level time-stamped traces into source-level timing analysis [13].

Stattelmann and Martin propose a solution for non-intrusive, context-sensitive tracing of defined program fragments [14]. They describe how programmable trigger logic can be used to overcome the technical limitations imposed by small trace buffers found in simple on-chip debugging solutions, arguing that context information can reduce pessimism during the composition of local execution times.

Li et al. describe an extension of IPET for modeling direct mapped instruction caches. In their approach the variables for individual nodes are split by cache line and hit/miss condition, opening the possibility to associate different costs with each case [26], [27]. By analyzing the *cache conflict graph*, a safe overapproximation of potential cache conflicts, they are able to derive constraints that limit the number of cache misses. Function calls are inlined similarly to [24]. Later, the approach is extended to set-associative caches by analyzing *cache state transition graphs* [28], [29]. According to [24], the approach has scalability issues.

The latter approach addresses specific hardware. The behavior has to be modeled explicitly for the particular hardware on hand, whereas our approach is generic. Our only premise is that the temporal effects are at least partially exposed by the control flow.

## VII. Conclusion

The accuracy of measurement-based WCET estimation is influenced by two opposed effects: Optimism, which arises due to the coverage limit of the timing-relevant computer state (TRCS) of measurement, implies a potential underestimation of the global WCET. On the other hand, pessimism, which arises due to abstraction, implies a potential overestimation of the global WCET.

As our first contribution, we have presented strong evidence that the control flow decisions taken just before some program part is executed can expose a substantial part of the TRCS. Based on this evidence, we have developed our idea of using the TRCS information that is exposed through individual control flow decisions to reduce pessimism.

As our second contribution, we have described how this idea can be integrated into IPET-based WCET calculation.

Lastly, we have presented confirmative experimental results demonstrating that the presented context-sensitive approach can significantly reduce pessimism in measurement-based WCET estimation.

## Acknowledgment

The research leading to these results has received funding from IST FP-7 project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering (ADVANCE)" under contract no IST-2010-248828 and from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within project "Formal Timing Analysis Suite of Real-Time Systems" (FORTAS-RT) under contract P19230-N13.

## References

[1] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *3rd Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'08)*, ser. Communications in Computer and Information Science, vol. 17, no. 8. Springer, Oct. 2009, pp. 430–444.

[2] R. Kirner, A. Kadlec, and P. Puschner, "Precise worst-case execution time analysis for processors with timing anomalies," in *21st Euromicro Conference on Real-Time Systems*. IEEE, Jul. 2009.

[3] P. Puschner and A. Schedl, "Computing maximum task execution time - a graph-based approach," *Journal of Real-Time Systems*, vol. 13, no. 1, pp. 67–91, Jul. 1997.

[4] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, Dec. 1997.

[5] H. Kopetz, *Real-time systems*, ser. The Kluwer international series in engineering and computer science. Boston, Mass.: Kluwer, 1997.

[6] R. Kirner and W. Haas, "Automatic calculation of coverage profiles for coverage-based testing," in *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Oct. 2009.

[7] R. Kirner, "Towards preserving model coverage and structural code coverage," *EURASIP Journal on Embedded System*, 2009.

[8] R. Kirner, P. Puschner, and A. Prantl, "Transforming flow information during code optimization for timing analysis," *Journal of Real-Time Systems*, vol. 45, no. 1-2, pp. 72–105, Apr. 2010.

[9] A. Holzer, M. Tautschnig, H. Veith, and C. Schallhart, "How did you specify your test suite?" in *25th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE'10)*, Sep. 2010, pp. 407–416. [Online]. Available: http://portal.acm.org/citation.cfm?id=1858996.1859084

[10] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, "Improving the confidence in measurement-based timing analysis," in *14th IEEE Intl. Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)*, Mar. 2011.

[11] S. Bünte, M. Zolda, and R. Kirner, "Let's get less optimistic in measurement-based timing analysis," in *6th IEEE Intl. Symposium on Industrial Embedded Systems (SIES'11)*, Jun. 2011.

[12] "Powertrace," Product Information from Lauterbach GmbH, Höhenkirchen-Siegertsbrunn, Germany. [Online]. Available: http://www2.lauterbach.com/doc/powertrace.pdf

[13] A. Betts, N. Merriam, and G. Bernat, "Hybrid measurement-based WCET analysis at the source level using object-level traces," in *10th Intl. Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 54–63. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2825

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

[15] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[16] M. Zolda, S. Bünte, and R. Kirner, "Towards adaptable control flow segmentation for measurement-based execution time analysis," in *17th Intl. Conference on Real-Time and Network Systems (RTNS'09)*, Oct. 2009.

[17] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks - past, present and future," in *10th Intl. Workshop on Worst-Case Execution Time Analysis*, Jul. 2010.

[18] F. Nemer, H. Cassé, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, "Papabench: a free real-time benchmark," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2006.

[19] Infineon, *TriBoard TC1796 Hardware Manual*, http://www.infineon.com, 2005.

[20] ——, *TC1796 User's Manual V2.0*, http://www.infineon.com, 2007.

[21] R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Cassé, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, M. Islam, R. K. Daniel Kästner and, L. Kovács, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr, "Wcet tool challenge 2011: Report," in *11th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, and R. Wilhelm, Eds., 2011, to appear.

[22] 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.

[23] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "An introduction to test specification in FQL," in *Haifa Verification Conference (HVC 2010)*, ser. Lecture Notes in Computer Science, S. Barner, D. Kroening, and O. Raz, Eds., vol. 6504, 2011, pp. 9–22.

[24] H. Theiling and C. Ferdinand, "Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis," in *RTSS '98: IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1998, p. 144.

[25] A. Betts and G. Bernat, "Tree-based WCET analysis on instrumentation point graphs," in *9th IEEE Intl. Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*. IEEE Computer Society, 2006, pp. 558–565. [Online]. Available: ttp://dx.doi.org/10.1109/ISORC.2006.75

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

[27] Y.-T. S. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 3, pp. 257–279, 1999.

[28] ——, "Cache modeling for real-time software: beyond direct mapped instruction caches," in *Real-Time Systems Symposium, 1996., 17th IEEE*, dec 1996, pp. 254 –263.

[29] ——, "Efficient microarchitecture modeling and path analysis for real-time software," in *RTSS '95: 16th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1995, p. 298.