Accurate Measurement-Based WCET Analysis in the Absence of Source and Binary Code

Amine Marref
Department of Computer Science
Umm Al-Qura University
Makkah, Saudi Arabia
ajmarref@uqu.edu.sa

Adam Betts
Mälardalen University
School of Innovation, Design, and Engineering
Västerås, Sweden
adam.betts@mdh.se

***Abstract*—Estimating the worst-case execution time (WCET) of real-time embedded systems is compulsory for the verification of their correct functioning. Traditionally, the WCET of a program is estimated assuming availability of the program's binary which is disassembled to reconstruct the program, and in some cases its source code to derive useful high-level execution information. However, in certain scenarios the program's owner requires that the binary of the program not be reverse-engineered to protect intellectual property; and in extreme situations, the program's binary is not available for the analysis, in which case it is substituted by program-execution traces.**

In this paper we show that we can obtain WCET estimates for programs based on runtime-generated or owner-provided time-stamped execution traces and without the need to access the source code or reverse-engineer the binaries of the programs. We show that we can provide very accurate WCET estimations using both integer linear programming (ILP) and constraint logic programming (CLP). Our method generates safe and tight WCET estimations for all the benchmarks used in the evaluation.

I. Introduction

Real-time systems (RTSs) must operate in a timely manner to ensure their correct functioning. A RTS is a set of tasks that cooperate in order to deliver a specific functionality. To ensure that the RTS works correctly, a schedulability analysis is performed which checks whether or not all tasks can meet their deadlines at runtime – this requires knowledge about the worst-case execution time (WCET) of the individual tasks. In order to estimate the WCET of a task, some analysis and/or testing is performed – which is usually classified like the following [1].

First, there is WCET estimation based on end-to-end measurements of the program. In this case, the program is executed for a set of input vectors generated through some test-data generation technique. After this, the largest-observed execution time during testing is augmented by some amount derived by engineering wisdom gained from experience testing similar programs. The augmented largest-observed execution time is then considered to be the WCET.

Second, there is static WCET analysis which creates mathematical models of the program and the hardware platform. In this case, the binary of the program is decomposed into building units – usually basic blocks, after which the timing behaviour of each building unit is derived based on the created hardware model. Some optional analysis of the program code might be performed to derive valuable functionality information that helps obtain more accurate WCET estimates. The timing behaviour of the building units together with knowledge about their structure in the program are used to derive the WCET of the program.

Third, there is (hybrid) measurement-based WCET analysis which creates a mathematical model of the program and runs its binary on the actual hardware. In this case, the program code is instrumented, then run for some test vectors generated using some test-data generation technique. The result is a set of time-stamped program traces that – together with the flow information obtained from program-flow analysis – are used to derive the WCET. An alternative to program instrumentation is the use of a hardware debug interface to automatically collect time stamps at predefined program locations e.g., at program branches.

Our objective in this work is to estimate the WCET of a program under the constraint that no reverse-engineering of the binary is allowed. One example is from the component-based development of real-time embedded software, where a software component can potentially be shipped with the restriction that no reverse-engineering of its binary is allowed. Another example is highly-confidential code where not even the binary program can leave the development site – in which case the analysis has only access to a very limited and minimum format of execution traces.

This immediately rules out static analysis since it requires the ability to decompile the binary or compile the source so as to obtain the instructions inside basic blocks (for pipeline, instruction/data cache, and branch-prediction analysis). The use of end-to-end measurements to obtain the WCET, on the other hand, does not require reverse-engineering the binary. However, since it is based on engineering wisdom, it lacks scientific grounding and is therefore unsuitable in the general case.

This leaves us to explore what (hybrid) measurement-

based analysis has to offer in terms of solving the problem that we address in this paper.

The contributions of the paper are the following.

- We show an analysis method to derive the WCET of programs based on information derived from traces only.
- We show that by performing or requiring a non-expensive testing of the program based on branch coverage together with careful time stamping, we are able to reconstruct the exact control-flow graph of the program that is completely unaware of the semantics of the program and hence comply with the non reverse-engineering requirement.
- We show how to derive timing constraints from time-stamped execution traces which are used to derive a tight WCET estimation using integer linear programming (ILP) and constraint logic programming (CLP) calculations.
- We show that using automatic test-data generation based on evolutionary search, we cover enough execution times of the program that enable the derivation of safe and tight WCETs for the benchmark programs used in the evaluation.
- We make a comparison of our estimated WCETs against the real WCETs obtained via exhaustive path coverage and those obtained using a combination of state-of-the-art program-flow and processor-behaviour static-analysis tools.

The rest of the paper is structured as follows. In Section II, we review related work in (hybrid) measurement-based analysis (MBA) and discuss why it cannot solve the problem that we address in this paper. In Section III, we formally define the problem of estimating the WCET based on time-stamped traces only. In Section IV, we describe how we generate time-stamped traces at the producer site or at the user site in the case they are not readily available to the MBA analysis. In Section V, we show the way by which our MBA uses the program-execution traces in order to derive the necessary timing information for the WCET estimation. In Section VI, we explain how use the derived timing information in our WCET estimation step. In Section VII, we compare the timing estimations that we obtain using our MBA approach against those obtained via SA and exhaustive path testing. Finally, in Section VIII, we include concluding remarks and directions for future research.

II. Related Work

A comprehensive survey on WCET analysis is in [1]. The most relevant comparison metric would be to review how WCET-analysis methods in the literature estimate the WCET given the restrictions of non-availability of source code and prohibition of reverse-engineering the binary. It turns out that all works violate at least one of these conditions. In this section we shall exclusively review MBA works where we focus on highlighting the key differences in estimating the WCET using our MBA and state-of-the-art MBA. A review of SA techniques for WCET estimation is not of relevance as they are very different.

Williams [2] employs a form of limited path coverage using *PathCrawler* [4] to find the WCET of the program – which suffers from obvious scalability problems. In our MBA, we avoid scalability issues by computing local execution times of program segments, deriving bounds on their execution counts, and combining this information globally to obtain the WCET.

Wegener et al. [5], Khan and Bate [6] use genetic algorithms to trigger long end-to-end execution times. We also use genetic algorithms to trigger long execution times during testing program segments; however, the WCET estimates that we compute are not the end-to-end observed execution times.

Bernat et al. [8] use probabilistic MBA where probability distributions of execution times of program segments are created and combined to form the overall WCET. In our case, we collect constraints on observations of execution times as opposed to create probability distributions of them.

Wenzel et al. [9] decompose the program into segments where path coverage is feasible, perform the local path coverage, and combine the largest execution times of the segments to obtain the overall WCET. We do not attempt local path coverage as it is prohibitively expensive – Wenzel et al. circumvent the problem by restricting their analysis to programs without loops.

Stattelmann [10] uses a MBA where the execution times of program segments are collected through an SA-controlled measurement process. The maximum (observed) execution times for segments are used to obtain the overall WCET of the program. In our MBA, we distinguish between the different execution times of program segments by deriving their maximum number of occurrences per trace.

Betts [11] performs MBA using a data structure called the Instrumentation Point Graph (IPG), which is derived from the locations of instrumentation with respect to program structure. However, he has only shown how to construct the IPG when the source code or disassembly is available; therefore it is unable to handle the case where the only input is execution-time program traces. In contrast, the technique presented in this paper builds the graph on the fly during trace processing.

More recently, Marref [12] has presented predicated WCET analysis, which deduces execution time effects between basic blocks from timing traces. These execution time effects are later injected into the calculation as constraints on execution paths, consequently tightening the WCET calculation. Once more, his method differs in that basic blocks serve as the unit of computation and, therefore, the CFG is the underlying program model which is constructed from the disassembly.

III. Problem Description

The problem for which we offer a solution in this paper is captured by the following two questions: (i) given that a real-time program is shipped by the producer with the restriction that it cannot be reverse-engineered, how one might derive a WCET estimate for it? (ii) if both source and binary files of the program are not available but time-stamped traces of its execution are available, how can one deduce the WCET from the traces alone?

We first address question (i). The term reverse-engineering in the context of this paper means any processing activity applied to the binary file of the program, and from which one can infer information about the semantics of the code. This includes for example using a disassembler that rebuilds the instructions of the program which disclose its algorithms.

We need to carefully define what is acceptable MBA practice under the restriction of the non reverse-engineering requirement. This will be any processing activity involving the binary of the program that does not reveal the semantics of the program. The interaction of the MBA with the binary program will be to execute it for different input vectors and to collect the resulting time-stamped traces which should not reveal any information about the program that could infringe the reverse-engineering restriction. In a subsequent stage, MBA builds the control-flow graph (*CFG*)[1] of the program based on the observed traces that we call the observed control-flow graph (oG); which is used in a later calculation stage to derive the WCET. Consequently, MBA can be used to solve question (i) if the generated traces together with their associated oG do not reveal program semantics at a level that violates the non reverse-engineering requirement.

Let us consider the following scenario. In order to perform our MBA, we could use for example a profiling device that reads all instructions being executed in the CPU at runtime (e.g., a logic analyser) and stores some description of them in a trace file. Assume that the (hypothetical) profiling device is able to identify the types of instructions being executed. The generated trace file reflects a complete execution of the program; and if used together with the associated oG, one can reconstruct the program under analysis – or at least its parts that executed. This clearly violates the non reverse-engineering requirement. This leads us to identify an important requirement of our MBA analysis: it should not reveal the type of instructions being executed.

Let us assume that we can set the profiling device to conceal the type of instructions being executed and to merely generate sensible identifiers for them. These identifiers could be memory locations for example; in which case the nodes of the oG will be uniquely identified by memory addresses connected via edges. Here, each node of the oG contains a sequence of memory addresses and their associated time stamps. The only semantics information that can be inferred from the oG in this case is that the last memory address of each node corresponds to a branch instruction. The MBA in this case reveals that the program under analysis contains a specific number of branches and their exact memory addresses. We believe that revealing such information does not violate intellectual property. The requirement on the MBA is then updated to: it should conceal all types of instructions except branches which are implicitly and unavoidably identified from the constructed oG.

As an answer to question (i), a MBA which satisfies the non reverse-engineering requirement is based on the analysis of time-stamped traces that only contain identifiers to (a subset of) program instructions together with their time stamps. Memory addresses are good candidates for such identifiers as they uniquely identify program instructions. This uniqueness property is required in order to construct the semantics-unaware oG. The addresses of instructions can be replaced by arbitrary identifiers as long as they maintain the uniqueness property. This is particularly useful in case where the size of the program – which can be inferred from the addresses of the first and last executing instructions – is also intellectual property. The answer to question (ii) is related to the previous discussion: as long as the program's owner provides program traces in the format that we have defined, we can perform a WCET estimation.

In summary, the problem of performing MBA analysis on a program whose source code is not available and binary be not reverse-engineered or substituted with time-stamped traces as described; reduces to generating the time-stamped traces – at producer or user site – in a manner that does not reveal program semantics; and making use of these traces in generating the WCET.

IV. Generating Time-Stamped Traces

The input to the MBA is a set of time-stamped traces generated at either the producer or user sites. In Section III, we have reached the conclusion that the time-stamped traces should contain unique identifiers and their corresponding time stamps.

In order to obtain time-stamped execution traces, one can use software instrumentation, hardware instrumentation, or a mixture of both. One problem with software instrumentation is that it is intrusive (also known as the probe effect) and will affect the timing behaviour of the program depending on the used instrumentation profile (lightweight or heavyweight). Another problem with software instrumentation – in this research context – is that it can only be applied by the software producer since the user does not have access to the source code.

On the other hand, hardware instrumentation through the use of hardware-debug interfaces has the advantage that it

[1]The nodes of a CFG are basic blocks, which are sequences of instructions in which flow of control can only enter at the beginning and leave at the end.

allows us to bypass the need to perform intrusive instrumentation to the software, and can conveniently be used at both user and producer sites. The problem with hardware instrumentation is that the CPU where the program runs must support time stamping by generating own time stamps or having available pins to connect to an external debug interface. Another problem with the use of debug interfaces is the data blackout phenomenon where the tracing process cannot keep-up with the speed of the CPU in executing instructions.

The *Nexus 5001* interface [13] is an example debug interface that can be used for this purpose as it can generate time stamps for program instructions that cause a change in flow – most importantly branch instructions. The generated trace is a sequence of pairs where each pair is the full logical address of the branch instruction together with a time stamp which can either be absolute or relative to previous time stamps. The tracing mechanism incorporates a self-correcting mechanism by which incorrectly-taken branches – by speculative execution – are not reported in the trace. The *Embedded ARM Macrocell* [14] in ARM processors also generates time-stamped execution traces of branch instructions.

In this work, we shall not restrict ourselves to a particular execution-tracing method. We assume that the traces have been generated and are ready for the MBA to use. The objective of this section is to show that *there are* ways to generate the input to our MBA method.

In summary, in order to obtain time-stamped traces for our MBA, we can either require the producer to instrument their software, or rely on hardware to generate time-stamped traces. The latter is preferred due to its non-intrusiveness; and in the case where the producer cannot generate the time-stamped traces – they can be generated at the user site.

V. The Measurement-Based Analysis

In this section, we shall discuss our MBA which is based on parsing time-stamped traces. The MBA either accepts the time-stamped traces as input or generates them in an initial step depending on the availability of the binary file. In this section, we assume that we have already generated or acquired the time-stamped traces and focus on the internals of the MBA step.

The input to our analysis is a set T of m traces T_i that correspond to the input vectors used during testing/measurements. We define a time-stamped program-execution trace T_i as a sequence of $|T_i|$ pairs $(p_j, st_j^h)_i$ where each pair is a program-point identifier p_j and its observed time stamp st_j^h at position $h, 0 \leq h < |T_i|$ in the trace T_i[2]. Each program point will be a node in the constructed oG. The nature of the program point depends on the way the traces are generated; a program point could for example be the first instruction or the last instruction of a basic block in the program. Time-stamping can occur at different pipeline stages e.g., at instruction fetch or commit stages.

We expect that the traces which are input to our analysis have a uniform way of choosing program points e.g., either the start or end of basic blocks but not a mixture; and we also expect a uniform way of time stamping them e.g., at the same pipeline stage.

After reading the traces, the MBA performs two intertwined steps that we describe here separately for the sake of clarity, namely building the oG and collecting execution time constraints.

A. Building the oG

In order to build the graph $oG = (V, E)$, the traces are parsed in order to identify the nodes V and the edges E between them. Each program point in the trace *is* a node in oG and any two adjacent pairs $(p_j, -)_i$ and $(p_{j'}, -)_i$ are *potentially* an edge in oG. The reason for this is that in the presence of speculative execution, a wrongly-fetched program point appears before a correctly-fetched program point in a trace, giving the impression that there is an edge between them. As a result, the derived oG includes extra edges compared to the original CFG. If the time stamps are generated at the instruction-commit stages instead, wrongly-fetched instructions do not commit and hence cause no extra edges in the oG.

The problem of generating an oG which is a superset of the original CFG can also be observed when the basic blocks are small (e.g., single-instruction blocks) compared to the depth of the pipeline. In this case, the instruction fetch of several contiguous basic blocks occur giving the impression that they execute in sequence and which yield several extra edges in the oG. This will hold with large blocks in superscalar pipelines since the instruction fetch of many blocks can start at the same time, giving – again – the impression that they execute in sequence. One way of eliminating the problem of obtaining superset graphs oG is to require that the time stamps are generated for commit stages of instructions.

There is also the problem of obtaining a subset graph oG. For example, a particular block in the program has never been exercised during testing then the constructed oG cannot contain this block. Another scenario is that an edge between two blocks has never been exercised during testing, in which case the resulting oG will not contain this edge. A solution to this problem is ensure that testing/measurements satisfy branch coverage [3] i.e., all branches and statements in the program have been exercised at least once during testing.

In summary, the oG is built on-the-fly by parsing traces. If testing satisfy branch coverage, and the time stamps correspond to the commit stages of instructions, the constructed oG will have the same structure as the original CFG. If

[2]The trace T_i is regarded as an array indexed from 0 to $(|T_i| - 1)$.

branch coverage is not satisfied, the CFG can potentially be a subset of the original CFG. The implication of this on the analysis is that the missing nodes/edges can contribute to the longest path in the program i.e., some optimism is introduced in this case. If time stamps do not correspond to commit stages, extra edges are identified in the oG which form longer (infeasible) paths that cause pessimism in the WCET estimation later on.

B. Collecting Time Constraints

Before performing timing analysis based on the traces, we first change the time stamps from *absolute* to *relative*. Each time stamp st_j^h represents the (absolute) moment in time when the program point executed. For MBA, we are interested in the execution time c_j^h of each node p_j of the oG which is delimited by the execution of its predecessor and successor nodes. The computation of c_j^h is trivial.

The (processed) time-stamped traces reveal very valuable information about the timing behaviour of the program. In particular, they show the frequency by which the different execution times of program points are observed. The frequency of observing a particular execution time is transformed into a bound during calculation. This can for example be useful in differentiating between the execution times of program points in the first iteration of loops and their execution times in subsequent iterations – for better accuracy of WCET estimation. The execution-time frequency information is translated into timing constraints later in the calculation stage.

Algorithm 1 shows how the execution times c_j^h of program points p_j are collected together with their associated frequencies. For each program point p_j, we define a vector t_j that stores the execution times c_j^h and a corresponding vector f_j that stores their frequencies ($|t_j| = |f_j| = \theta_j$).

Algorithm 1 Collecting the execution-time frequencies of program points p_j from the traces T.

Require: A set T of m traces T_i
Require: The set V of the nodes of oG
1: $T = \{T_i \bullet i \in [1..m]\}$
2: $\forall T_i \in T \bullet T_i =< \ldots, (p_j, c_j^h)_i, \ldots >$
3: $\forall p_j \in V$ define (empty) vectors $t_j[]$ and $f_j[]$
4: **for all** $T_i \in T$ **do**
5: **for all** $(p_j, c_j^h)_i \in T_i$ **do**
6: define empty vector $f'_j[]$
7: **if** $c_j^h \notin t_j$ **then**
8: $t_j \leftarrow t_j \cup [c_j^h]$
9: $f'_j \leftarrow f'_j \cup [1]$
10: **else**
11: $h \leftarrow$ position of c_j^h in t_j
12: $f'_j[h] \leftarrow (f'_j[h] + 1)$
13: **end if**
14: **end for**
15: $f_j[] \leftarrow \max(f_j[], f'_j[])$
16: **end for**

The objective of Algorithm 1 is to define for each program point p_j its vector t_j of execution times and its vector f_j of their maximum number of occurrences in any *one* trace. For each trace T_i, at the end of executing the inner loop of Algorithm 1, the auxiliary vector f'_j contains the maximum number of observations in trace T_i of every execution time recorded so far in the traces T. At the end of parsing all traces T, the vector f_j contains the maximum number of occurrences of every execution time observed in T at any one trace.

More than just deriving frequencies of occurrences of execution times, we can relate them to past execution. For example, some program point p_1 might exhibit two execution times $c_1^1 = 10$ and $c_1^2 = 20$ in a subset of the time-stamped traces. It could be the case that the execution time $c_1^2 = 20$ is observed only when another program point p_2 executes before p_1 and causes it to have the execution time $c_1^2 = 20$ e.g., because of cache interaction. Accounting for cases like this improves the accuracy of the WCET estimation especially in the case of large execution times that occur because of pathological interactions in hardware accelerators.

In order to collect more context-sensitive timing constraints, we parse the traces and identify for every execution time c_j^h of program point p_j, the set of program points p_k that must execute to observe the execution time c_j^h. The problem reduces to intersecting the traces where c_j^h is observed and the common program points are those that cause c_j^h to be observed. Associating sets of program points with particular execution times has two major challenges. On the one hand, it requires expensive trace parsing; and on the other hand, it is difficult to reason about which program points are causing which execution times (of other program points) inside loops because of the circular execution.

We conservatively overcome the two challenges by limiting the analysis to outside loops and performing it within a trace window i.e., when traces are intersected, only up to ω program points that executed in the past – prior to execution of p_j – are compared for common program points p_k. Algorithm 2 shows the way the context-sensitive execution time constraints are collected. The algorithm derives for every execution time c_j^h, the set A_j^h of program points that must execute – in traces T – in order to observe c_j^h.

Algorithm 2 is heuristic, it does not guarantee that the sets A_j^h of program points are exact, they can be subsets or supersets.

VI. Estimating the WCET

In order to estimate the WCET, the program flow and timing information are combined in a calculation stage which classically is path-based, tree-based, or IPET-based (IPET for implicit path-enumeration technique). In a path-based calculation, all the paths in the program or inside some program parts (e.g., loops) are enumerated; which does not scale in the general case. Tree-based calculations are based

Algorithm 2 Collecting the context-sensitive timing constraints of program points p_j from the traces T.

Require: A set T of m traces T_i
Require: The set V of the nodes of oG
Require: The vectors $t_j[]$ of program points p_j
for all $p_j \in V$ **do**
  for all $c_j^h \in t_j[]$ **do**
    define set $A_j^h = V$
    define empty set B_j^h
    for all $T_i \in T$ **do**
      if p_j occurs in T_i with time c_j^h **then**
        $B_j^h \leftarrow$ the previous ω program points of T_i not in a loop
        $A_j^h \leftarrow A_j^h \cap B_j^h$
      end if
    end for
  end for
end for

Table I
THE IPET MODEL OF THE MBA ANALYSIS BASED ON NON CONTEXT-SENSITIVE TIMING CONSTRAINTS.

1	$(ipred(p_j) = \emptyset) \Rightarrow (x_j = 1)$
2	$\forall p_j, \forall p_{k_1} \in ipred(p_j) \bullet x_j = \sum x_{k_1 j}$
3	$\forall p_j, \forall p_{k_2} \in isucc(p_j) \bullet x_j = \sum x_{j k_2}$
4	$\forall p_j \bullet x_j = \sum_{h=1}^{\theta_j} x_j^h$
5	$\forall p_j, \forall h \in [1..\theta_j] \bullet c_j^h = t_j[h] \wedge \hat{x}_j^h = f_j[h]$
6	$\forall p_j, \forall h \in [1..\theta_j] \bullet x_j^h \leq \hat{x}_j^h$
7	$\forall p_j, \forall h \in [1..\theta_j] \bullet x_j \leq \sum_{h=1}^{\theta_j} \hat{x}_j^h$

on abstract-syntax trees of programs which in our case can be constructed from the oG. A tree-based calculation is fastest given constant execution times of program points; but in our work a program point can potentially have multiple execution times which forces us to tweak the tree-based calculation to account for execution-time variability. We prefer to use the IPET approach where the timing variability of program points is transparently expressed as bounds on some of the variables, and is based on off-the-shelf linear and constraint programming solvers.

The program is represented as a graph circulation problem as shown in [15]. Each node p_j in oG has an execution count x_j and an execution time c_j. An edge between nodes p_{j_1} and p_{j_2} has execution count $x_{j_1 j_2}$. When program point p_j has θ_j observed execution times, its execution count x_j is split amongst θ_j execution counts $x_j^h, 1 \leq h \leq \theta_j$ corresponding to θ_j observed execution times c_j^h. The variable c_j^h is the magnitude of the observed execution time, and x_j^h is its execution count in the IPET model. The execution counts x_j^h are bounded by their maximum frequencies $\hat{x}_j^h$ in vector f_j constructed by Algorithm 1.

Table I shows the IPET model that we use in order to estimate the WCET based on non context-sensitive timing constraints. The variable $ipred(p_j)$ (respectively $isucc(p_j)$) is the set of immediate predecessors (respectively successors) – in oG – of program point p_j. The first three rows of Table I are the flow-preservation constraints, and the last four rows are the timing constraints together with their corresponding execution-count bounds. We do not use functionality constraints e.g., same-path program points or mutually-exclusive program points as they do not fall under the scope of this work.

In order to integrate context-sensitive timing constraints, we transform the sets A_j^h derived by Algorithm 2 – and which satisfy $|A_j^h| > 0$ – to time constraints according to Equation (1).

$$(\bigwedge(x_j > 0)) \Leftrightarrow (c_j = c_j^h), \text{ for all } p_j \in A_j^h \qquad (1)$$

We shall call the MBA based on the IPET model that corresponds to Table I augmented with context-sensitive timing constraints *the path-sensitive MBA* and the MBA based on the IPET model that corresponds to Table I only *the path-insensitive MBA*.

VII. EVALUATION

In order to evaluate the goodness of the WCET estimations using our MBA, we compare them to estimations provided by a static analysis tool to reason about safety, and to the *real* WCET to reason about tightness.

A. Experimental Setup

We apply our MBA analysis to a set of benchmark programs [16] which are traditionally used for evaluating WCET-analysis results.

We have modified the set of benchmark programs in order to make them amenable to MBA. First, the "main" function of each program is changed to accept inputs as arguments since most of the benchmark programs have their inputs hard-wired internally. Second, we had to make variable a subset of constant values in the benchmark programs – mostly loop induction variables – in order to increase the number of execution paths in them. Third, we have limited the input space of some benchmark programs to make exhaustive input-vector exploration feasible and by consequence make path coverage feasible.

In order to generate test data for our MBA, we use a combination of branch coverage and evolutionary test-data generation. We use *crest* [17] a concolic-execution [18] tool that automatically generates test vectors that cover branches in the program. We also use test-data generation based on genetic algorithms in order to cover the (small number of) branches that are missed by branch-coverage testing and also to trigger long execution times of the program under test. The fitness function that we use for the genetic algorithm is the total execution time of the program during runs; and for all benchmark programs, we use a fixed evolution of 100

generations with a population of 100 individuals in each generation.

In order to obtain the time-stamped traces, we use the *simplescalar* cycle-accurate simulator [19] to produce a "NEXUS-like" set of traces. There are many reasons that justify using a simulator for evaluation in our case. First, the central idea of this work is estimating the WCET based on time-stamped traces i.e., the source of the traces is orthogonal. Second, by writing a simple trace parser for the simulator, we can transform the rich simplescalar trace to a sequence of pairs where each pair is the commit stage of each branch instruction and its associated cycle time. In this case, the trace – after parsing – looks like the one provided by a hardware debugger like Nexus in the sense that only branch instructions which commit in the pipeline are traced. A direct result in this case is that the oG that we derive is the same one that we derive if we were to use the Nexus hardware debugger.

The disadvantage of using a simulator like simplescalar is that it only models the main hardware accelerators e.g., pipelines, caches, and branch predictors. The timing behaviour of other hardware features, e.g., system bus is not accounted for. However, this disadvantage is in our favour when it comes to comparing our results to a static analysis tool which only models the main hardware features. In other words, comparing timing results obtained on hardware to static analysis results where only a subset of this hardware is modelled may not yield accurate conclusions.

The most appropriate static WCET analysis tool to use in our comparison is the *Chronos* timing analyser [20]. One reason to choose this tool is that it is made freely available for research purposes. The second reason is that it creates automatic hardware models from simplescalar configuration files. This way, we can define a hardware configuration to use in our simulations which will be the same one used by Chronos to derive the hardware model. The comparison of the WCET estimations between our MBA and a static analysis tool are very accurate in this case. Chronos requires the user to input loop bounds which we provide using *Sweet* [21] on an intermediate format generated from the C file. Notice that Chronos needs to disassemble the binary of the program and Sweet needs its source code which clearly violate the source non-availability and binary non reverse-engineering constraints.

B. Evaluation Framework

The evaluation is performed according to the framework depicted in Figure 1 which produces the three values: SA WCET estimation, MBA WCET estimation, and actual WCET.

SA WCET Estimation is derived using Chronos mainly with loop-bound annotations from Sweet. In Figure 1, this includes the steps: C program, (cross) compiler, binary, architecture configuration, Sweet tool, annotations, Chronos tool, and SA WCET estimation.

MBA WCET Estimation is derived using our MBA analysis. This is mainly the area labelled "MBA" in Figure 1. In order to obtain the input to the MBA, we generate time-stamped traces where the time stamps are created at the commit (pipeline) stages of branch instructions. In order to ensure that the oG is not a subset of the original CFG, we ensure branch coverage in testing using a combination of concolic execution and evolutionary testing. The trace-generation stage is labelled "Trace Generation" in Figure 1: the binary of the program is simulated on the cycle-accurate simulator by consuming the input vectors generated by the "the test suite for branch coverage and evolutionary testing" and following the edges labelled "BE" (branch coverage, evolutionary) in the trace generation box.

Actual WCET is derived by exhaustively executing the binary for the input vectors generated by the "test suite for path coverage" and following the edges labelled "PC" (path coverage) in the trace generation box. Notice that since we use simpescalar, we rely on parsing the trace in order to find the exact end-to-end execution time of the program.

It is clear from the framework in Figure 1 that the MBA only needs time-stamped traces as input. In this evaluation, branch coverage of the program is performed using the tool CREST which requires the source code to be available. In a realistic scenario, we delegate this task to the producer since only them have access to the source code. In this case, the input to our analysis is the binary together with input vectors that achieve the coverage that we want during the generation of time-stamped traces, or a ready-set of time-stamped traces shipped with the binary.

C. Results

We have applied the framework of Figure 1 on a subset of the benchmarks in [16]. If a benchmark program from [16] is not included in the evaluation, it is because either of Chronos or Sweet requires tedious user annotations or does not terminate within a practical time.

We have used a relatively complex hardware H with out-of-order pipeline, instruction cache, and dynamic branch prediction. The reason for not choosing a more complex hardware configuration than H is that Chronos takes an impractical time to solve the generated (heavily-constrained) ILP model for some benchmarks in Table II using *lp-solve* [22] – the free ILP solver. We do not account for data caches as Chronos does not support data-cache analysis.

We implement the path-insensitive MBA in ILP using *lp-solve*, and we implement the path-sensitive MBA in CLP using *eclipse* [23] the constraint logic programming solver. The reason for choosing CLP to implement the path-sensitive MBA is due to the nature of the context-sensitive timing constraints which are equivalences (or double implications)

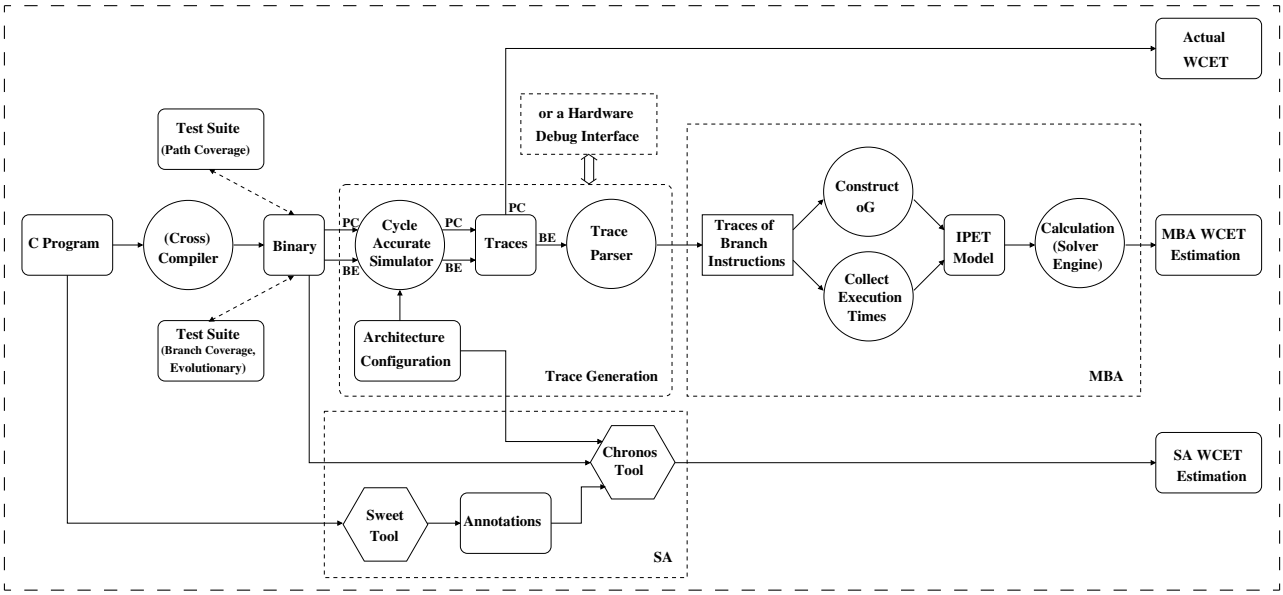

Figure 1. The evaluation framework.

Table II
WCET VALUES USING EXHAUSTIVE PATH-COVERAGE TESTING (W), OUR PATH-INSENSITIVE MBA BASED ON ILP (W_{MBA} USING ILP) WITH ASSOCIATED OVERESTIMATION (O_{MBA} USING ILP), OUR PATH-SENSITIVE MBA BASED ON CLP (W_{MBA} USING CLP) WITH ASSOCIATED OVERESTIMATION (O_{MBA} USING CLP), AND SA (W_{SA}) WITH ASSOCIATED OVERESTIMATION O_{SA} ON ARCHITECTURE H.

Program	W	W_{MBA} using ILP	O_{MBA} using ILP	W_{MBA} using CLP	O_{MBA} using CLP	W_{SA}	O_{SA}
binary_search	8554	9143	7%	8640	1%	15456	81%
bubblesort	37066	38157	3%	37102	0%	82298	122%
crc	1593775	1593775	0%	1593775	0%	4022840	152%
dct	495942	495954	0.0%	495954	0.0%	965081	95%
expint	6260	6592	5%	6420	3%	10769	72%
factorial	7556	8502	13%	8017	6%	15660	107%
fdct	503200	503200	0%	503200	0%	966755	92%
fibcall	58704	58828	0.0%	58790	0.0%	64833	10%
fir	5231162	6085089	16%	5728027	9%	7035550	34%
insertsort	32371	32623	1%	32435	0.0%	78741	143%
janne_complex	19091	21234	11%	19591	3%	60896	219%
matrix_cnt	138754	155609	12%	142303	3%	356086	157%
matrix_mult	893545	996928	12%	951205	6%	820041	-8%
select	94011	113123	20%	108966	16%	216956	131%

that cannot be handled in ILP without duplicating the ILP model or performing ad hoc transformations [24].

The results of the evaluation are shown in Table II. There is an underestimation by SA for one of the benchmark programs which is certainly due to a bug in either Sweet or Chronos since the theory underlying their analyses ought to produce safe estimates.

As can be seen from the Table II, our MBA outperforms SA for all the benchmarks. It is not the objective of this paper to dissect the sources of pessimism in the estimations of SA i.e., whether they relate to processor-behaviour analysis or program-flow analysis; or both.

All estimations by both MBAs are safe and the pessimism never exceeds 20% for path-insensitive MBA, and 16% for path-sensitive MBA.

D. Discussion

It is not possible to prove that our MBA produces safe WCET estimates in the general case, but it is trivial to show that it will always bound the largest-observed execution time. The MBA produces an unsafe WCET estimate when it fails to cover all iterations of some loops or when it fails to stress the maximum execution times of some program points.

The time and space required to perform our MBA mostly depend on the number and size of traces. For realistic-size programs, the trace-parsing overhead could be considerable.

The number of runtime measurements required in our MBA using both branch-coverage and genetic-algorithm testing are significantly smaller than those needed by path-coverage testing on average.

VIII. Conclusion

In this paper we have proposed and implemented a method for estimating the WCET of real-time systems based on time-stamped traces only. This is very important when the program under analysis comes with the restriction of non-reverse engineering the binary, or when it cannot be shipped to the analysis site; which is not uncommon.

The analysis is completely based on parsing traces to derive timing and execution count constraints on the observed program points in the time-stamped traces; which are then modelled as an ILP/CLP problem which by its turn is fed to an off-the-shelf ILP/CLP solver. The quality of the WCET estimation is influenced by the quality of testing, in particular coverage of the maximal execution times of the program points in the traces.

This work shows that it is possible to perform a completely out-of-the-box and automatic WCET analysis of real-time embedded software that outperforms end-to-end testing. It has the advantage of bounding the measured longest end-to-end execution time based on scientific analysis as opposed to using engineering wisdom. We believe that this increases the confidence of the user in the WCET estimations.

The work also shows that static WCET analysis is completely unable to solve the problem at hand – which is an increasingly emerging trend in the area of component-based embedded systems.

As a future work, we will look into more clever execution-time coverage by which the number of generated traces is decreased. We will also attempt to apply the technique on larger programs (subject to availability) using actual hardware debug interfaces.

Acknowledgements

This work is supported by the Swedish Foundation for Strategic Research (SSF) through the Research Centre for Predictable Embedded Software Systems (PROGRESS).

References

[1] 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 Transations on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] N. Williams, "WCET measurement using modified path testing," in *Proceedings of the 5^{th} International Workshop On Worst-Case Execution-Time (WCET) Analysis in conjunction with the 17^{th} Euromicro International Conference on Real-Time Systems*, 2005.

[3] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[4] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the 5^{th} European Dependable Computing Conference (EDCC'5)*, 2005, pp. 281–292.

[5] H. Pohlheim, J. Wegener, and H. Sthamer, "Testing the temporal behavior of real-time engine control software modules using extended evolutionary algorithms," *VDI BERICHTE*, vol. 1526, pp. 61–66, 2000.

[6] U. Khan and I. Bate, "WCET analysis of modern processors using multi-criteria optimisation," in *Proceedings of the 1^{st} International Symposium on Search Based Software Engineering (SSBSE'09)*, 2009, pp. 103–112.

[7] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proceedings of the 23^{rd} Real-Time Systems Symposium (RTSS'2002)*, Austin, Texas, USA, December 2002, pp. 279–288.

[8] G. Bernat, M. Newby, and A. Burns, "Probabilistic timing analysis: an approach using copulas," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 179–194, 2005.

[9] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Proceedings of the 3^{rd} International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, October 2008.

[10] S. Stattelmann, "Precise Measurement-Based Worst-Case Execution Time Estimation," Master's thesis, Saarland University, Faculty of Natural Sciences and Technology I, Department of Computer Science, September 2009.

[11] A. Betts, "Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs," Ph.D. dissertation, University of York, November 2008.

[12] A. Marref, "Predicated Worst-Case Execution-Time Analysis," Ph.D. dissertation, York, UK, 2009.

[13] The Nexus 5001 Forum, "Standard for a Global Embedded Processor Debug Interface: Version 2.0," *http://www.nexus5001.org/standard*, December 2003.

[14] ARM Information Centre, "Embedded Trace Macrocell ETMv1.0 to ETMv3.4 Architecture Specification," *http://infocenter.arm.com/help*, 2007.

[15] P. Puschner and A. Schedl, "Computing Maximum Task Execution Times - A Graph-Based Approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[16] Mälardalen WCET research group, "WCET project/benchmarks," *http://www.mrtc.mdh.se/ projects/wcet/benchmarks.html*, January 2010.

[17] J. Burnim, "CREST - automatic test-generation tool for C," *http://code.google.com/p/crest/*, April 2010.

[18] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/FSE-13: Proceedings of the 10^{th} European software engineering conference held jointly with 13^{th} ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: ACM, 2005, pp. 263–272.

[19] D. Burger and T. Austin, "The simplescalar tool set, version 2.0," University of Wisconsin, Madison, Technical Report CS-TR-1997-1342, 1997.

[20] X. Li, Y. Liang, T. Mitra, and A. Roychoudury, "Chronos: A Timing Analyzer for Embedded Software," *Science of Computer Programming*, vol. 69, no. 1-3, pp. 56–67, 2007.

[21] WCET-IDT-MRTC, "SWEET," *http://www.mrtc.mdh.se/ projects/wcet/sweet.html*, January 2010.

[22] M. Berkelaar, "lpsolve, version 5.5.15," *http://sourceforge.net/projects/lpsolve/*, March 2010.

[23] K. Apt and M. Wallace, *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press, 2007.

[24] A. Marref and G. Bernat, "Predicated Worst-Case Execution-Time Analysis," in *Ada-Europe 2009: Proceedings of the 14^{th} Ada-Europe International Conference on Reliable Software Technologies*. Berlin, Heidelberg: Lecture Notes in Computer Science (LNCS), Springer, 2009, pp. 134–148.