Abstract—Obtaining tight worst-case execution-time (WCET) estimations of real-time tasks is crucial since overly-pessimistic estimations are deemed impractical. One way of making WCET estimations tighter is to incorporate more program-flow information e.g., context-sensitive loop bounds, infeasible-path and same-path information, etc.

In this paper we present and evaluate a completely automatic analysis that dynamically derives program-flow information to use in WCET analysis. Flow information is derived by a combination of test-data generation and parsing of program-execution traces to obtain flow-fact hypotheses which are then fed to a model checker to establish their correctness. Experimental evaluation shows that our method help achieve considerable tightness in WCET estimations at a manageable cost.

I. INTRODUCTION

Real-time systems (RTSs) must operate in a timely manner to ensure their correct functioning. An 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, a program-flow model and a hardware-timing model are constructed, followed by a calculation of a WCET estimate [24].

A program-flow model (or program model) decomposes the program into (execution) units e.g., basic blocks; and describes their structuring, the way they are related to each other e.g., in a control-flow graph or an abstract-syntax tree, in which loops of the program they reside, etc. In addition to this, a program model optionally describes which units always execute together, which units never execute together, which units execute part of which execution context e.g., inside a loop nest or during a function call, etc. We call the information contained in the program model flow constraints.

A hardware-timing model (or a hardware model) describes the timing behaviour of the program units given some hardware architecture by taking into account hardware accelerators e.g., pipelines, caches, branch predictors, etc.

Finally, the program model and the hardware model are used in a calculation stage to produce a WCET estimate for the program under analysis. Such derived WCET estimate must be safe for hard RTSs or safe with a high level of confidence for soft RTSs, and should be tight.

The tightness of the WCET estimation is very important e.g., if an overly-pessimistic WCET estimate is fed to some schedulability analysis, it might cause the non-schedulability of the system; it might also generate a false warning that the program cannot fit in its timing budget. To ensure that the WCET estimation is reasonably tight, more execution context must be accounted for which can be performed at the program model or the hardware model.

In the program model, specifying that two program units have mutually-exclusive execution can potentially enhance tightness if they have large execution times: in this case, in the calculation stage they cannot both contribute to the WCET estimation at the same time. In a symmetric way, same-path information also benefits tightness. Relative execution is also beneficial e.g., some program unit executes four times for every execution of some other program unit.

In the hardware model, tightness is normally achieved by using more detailed timing information e.g., specifying the concrete execution times of some program unit given different hardware states where it executes instead of using one global (safe) execution time that relies on a pessimistic abstract hardware state. Seeking tightness this way is expensive as the task of estimating the WCET becomes hindered by an explosion of hardware states for different hardware accelerators.

For this reason, we focus on achieving tightness in WCET estimation at the program model. To make this possible, there must be a way to automatically derive the required program-flow constraints that restrict the program model to make the WCET estimation tighter. Unfortunately, there is no efficient way in the literature that enables the automatic derivation of accurate flow constraints without unrealistic assumptions about the input size of the program or its structure and semantics.

In this paper, we show a fully-automatic method to derive exact flow constraints based on a combination of test-data generation, concrete execution, hypothesis derivation, and model checking. The contributions of the paper are the following.

- A fully automatic method to identify exact flow constraints of a real-time task.
- A test-data generation suite for obtaining execution traces optimized for hypothesis derivation.

This research is funded by the program of strategic technologies of the national plan for science, technology, and innovation in Saudi Arabia No. 10-ELE1237-10.
• A hypothesis-generation algorithm that derives flow-constraint hypotheses based on concrete runtime-generated data.

The paper is structured as follows. Section II describes related work in the automatic generation of flow facts for WCET analysis. Section III describes the problem of generating flow facts dynamically and validating them. It explains the types of flow facts we generate, the method we use to generate them, and the way we validate them. Section IV conducts a thorough evaluation of our method. Section V summarises our work, derives conclusions, and sets directions for future work.

II. RELATED WORK

A comprehensive overview of WCET analysis is available in [24]. The WCET analysis includes three main steps: program-flow analysis, processor-behaviour analysis, and calculation. This work falls under flow analysis and so we shall not review processor-behaviour analysis or calculation.

Flow analysis for WCET analysis is used to derive mandatory program-execution information such as loop and recursion bounds — whenever possible — without which the WCET analysis reduces to the Halting problem [23]; and useful program-execution information which helps produce tighter WCET estimations because it restricts the execution of certain regions of the analysed program to always execute together, to never execute together, to occur a specific number of times, etc. Flow analysis has been performed both statically and dynamically in the literature.

Static flow analysis has been used for the automatic derivation of loop bounds by Healy et al. [9], Holsti et al. [10], Gustafsson et al. [8] using abstract interpretation, Bate and Kazakov [1] using inductive logic programming [17], Kebbal [11] using symbolic execution, and others. Static flow analysis has also been used for the detection of infeasible paths and analysis of input space for functions. Infeasible paths are detected by Chapman [5], Altenbernd and Kebbal [11], [21] using symbolic execution, Gustafsson and Ermedahl [8] using abstract interpretation, Stein and Martin [22] using constraint analysis of path-traversal conditions to list but a few. The research in static flow analysis is generally not mature enough to tackle real-life applications [14]. Code with non-structured loops causes major problems to static flow analysis, and code with triangular loops is not handled efficiently by state-of-the-art flow-analysis techniques as overly pessimistic bounds are usually generated for triangular loops.

Dynamic flow analysis has been performed by Prantl et al. [18], [20] to verify correctness of user-input loop bounds. This method resembles ours in the sense that it uses dynamic testing combined with model checking. However, our method is more general as it can derive a superior set of flow facts and incorporates test-data generation methods for obtaining good hypotheses that reduce the time needed for model checking.

III. PROBLEM FORMULATION

The objective of this work is to define a flow-constraint generation method that automatically derives exact flow constraints of some program which can be integrated in a WCET estimation. Traditionally, the WCET of a program is obtained by decomposing the program into \( n \) constituent units \( u_i \) arranged in a graph or a tree that describes their execution. Then the WCET is estimated by combining the execution times \( c_i \) and execution counts \( x_i \) of the units \( u_i \). The execution times \( c_i \) are determined by hardware timing analysis and their derivation is not the subject of this work. The WCET literature [24] is mostly dominated by research on how to derive the values \( c_i \). In this work, we are exclusively concerned with the variables \( x_i \).

Each program unit \( u_i \) of some program \( p \) has an associated execution count \( x_i \) which specifies the number of times \( u_i \) executes in any single run of \( p \). The output of the flow-constraint generation method is a set of flow constraints involving one or more program units \( u_i \).

Formally, a flow constraint is a predicate \( C(x_1, x_2, \ldots, x_n) \) that describes the execution behaviour of units \( u_1, u_2, \ldots, u_n \).

Table I shows example flow constraints. We shall describe the mathematical properties of the constraint \( C \) shortly.

In order to derive flow constraints we proceed (in a nutshell) as follows. First, program \( p \) is executed for a number \( m \) of input vectors yielding \( m \) execution traces that contain information about execution counts \( x_i \). The traces are parsed to build an \( m \)-by-\( n \) matrix \( X \) of the \( n \) execution counts \( x_i \) in the \( m \) runs.

Next, a hypothesis-generation technique — that we shall define later — takes matrix \( X \) as input and produces a set of hypothetical constraints \( \mathcal{C} \) (or hypotheses) over the variables \( x_i \) which capture their relationships in the matrix \( X \). For any set of one or more variables \( x_i \), there will be one or more hypotheses \( \mathcal{C} \) involving the set and which hold true for the set over all the rows of matrix \( X \).

After that, each hypothesis \( C \) is passed as an assertion together with the program under analysis to a model checker which either establishes its correctness or provides a counter example in the form of an input vector that causes program execution to yield an assignment to the variables \( x_i \) which violates the hypothesis.

Finally, for \( k \) derived hypotheses, a number of \( k_p \leq k \) hypotheses \( P \) that pass model checking are declared as correct flow constraints and added to the WCET analysis. Otherwise, the counter examples of the \( k_f \leq k \) failed hypotheses are used to generate a set of \( k_f \) new vectors that yield an updated \( X \) matrix of \( m \leftarrow m + k_f \) rows. The hypothesis-generation and validation stages are repeated until some stopping criterion is met.

We assume that the WCET analysis uses a calculation method that benefits from the derived flow constraints. Most

---

**TABLE I**

<table>
<thead>
<tr>
<th>#</th>
<th>Flow Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( C(x_1) \land x_1 \leq b_1 )</td>
<td>( u_1 ) executes at most ( b_1 ) times</td>
</tr>
<tr>
<td>2</td>
<td>( C(x_1, x_2) \land x_1 + x_2 = 1 )</td>
<td>( u_1 ) and ( u_2 ) are mutually exclusive</td>
</tr>
<tr>
<td>3</td>
<td>( C(x_1, x_2) \land x_1 = f_1 x_2 )</td>
<td>( u_1 ) executes ( f_1 ) times as often as ( u_2 )</td>
</tr>
</tbody>
</table>

201
WCET analyses nowadays base the calculation stage on the implicit-path enumeration technique (IPET) [13], [19], and so we confine our discussion on the flow-constraint generation to WCET analyses which estimate the WCET using IPET.

In the following, we conduct a more thorough discussion of the different aspects of the analysis: properties of hypotheses $C$, hypothesis generation, input-data generation, and hypothesis validation.

A. Properties of Hypotheses $C$

There are some attributes of hypotheses $C$ that ought to be taken into account when considering their derivation.

Usefulness. A derived hypothesis must be useful in terms of improving the WCET estimation. This can be seen from two angles.

First, the WCET analysis should make use of such hypothesis after its validation. For example, a hypothesis $C$ involving an exponential relation between some variables $x_1$ and $x_2$ in the form $C(x_1, x_2) \iff x_1 = a^{x_2}$ given some constant $a$ is hardly of any use. The reason for this is that non-linear constraints make the calculation of the WCET estimate harder, and in some cases impossible e.g., when the calculation is based on IPET modelled as integer-linear programming (ILP) [25] which cannot handle non-linear constraints. We shall focus on deriving linear hypotheses since modelling the WCET as in ILP-based IPET is the most common way of estimating the WCET in the literature.

Second, the derived hypotheses should make the WCET estimation tighter indeed. For example, a (validated) hypothesis of the form $C(x_1) \iff x_1 \geq 0$ is guaranteed not to tighten the WCET estimation since the IPET model already knows that all execution counts are non-negative integers. Unfortunately, there is no formal way to know whether or not a flow constraint will benefit the WCET estimation (except the previous example), and so the derived hypotheses $C$ might not tighten the WCET estimation. Any potential usefulness of adding the validated hypotheses to the WCET analysis can only be measured through empirical evaluation.

Mathematical Properties. The method we suggest involves deriving a hypothesis then proving its correctness. During the hypothesis-derivation process, the analysis is informed about the general shape of the hypothesis that ought to be derived. For example, whether the hypothesis is linear or not, whether it involves two variables or more, etc. It happens that the hypothesis-derivation process becomes more expensive when the hypothesis to be derived is more complex — we limit our derivation to simple hypotheses only.

Based on the previous informal discussion, we now define the hypotheses $C$ that we generate by our proposed analysis. We shall distinguish between them based on their arity (number of arguments), and their form (how they are expressed).

1) Unary Hypotheses: A unary hypothesis $C$ is expressed over one argument $x_i$ and has the form that corresponds to the rows 1-3 of Table II.

Row 1 indicates that unit $u_i$ always executes a specific number $b_i$ of times. This constrains the returned path by WCET analysis to include exactly $b_i$ occurrences of unit $u_i$ which could be beneficial. For example, if some units $u_1$ and $u_2$ are two alternatives of a conditional statement that executes inside a loop and satisfy $c_1 < c_2$ (remember that $c_i$ are execution times), then the WCET can potentially be reduced by an amount $(b_1(c_2 - c_1))$ because otherwise the IPET calculation returns unit $u_2$ with the superior execution time $c_2$ as executing all the way through the program.

Row 2 indicates that unit $u_i$ executes at least $b_i$ times. This could tighten the WCET, e.g., a program unit $u_i$ with a small execution time — which otherwise will not be returned as part of the longest path due to its small contribution to the WCET — is forced to be part of the longest path.

Row 3 indicates that unit $u_i$ executes at most $b_i$ times. This could tighten the WCET, e.g., a program unit $u_i$ with a large execution time is forced to be returned only $b_i$ times or less as part of the longest path.

In general, unary hypotheses $R$ narrow the domains of the values $x_i$ which has the potential of making the WCET estimation tighter. They also indirectly express the same-path property between units $u_i, u_j$ through the expressions of Rows 1, 2 if $b_i, b_j > 0$. For example if $x_1 = b_1, x_2 = b_2, b_1 > 0$, and $b_2 > 0$, then $u_1$ and $u_2$ are constrained to be on the same path since they both must execute at least once in the returned WCET path.

2) Binary Hypotheses: A binary hypothesis $C$ is expressed over two arguments $x_i$ and $x_j$ and has the form that corresponds to rows 4-7 of Table II. Rows 4-7 capture correlations in the execution of pairs of units $u_i$ and $u_j$.

Row 4 in particular expresses mutual exclusion between units $u_i$ and $u_j$ outside loops if $\alpha_{ij} = 1$ or inside loops if $\alpha_{ij} > 1$.

Rows 5 and 6 tighten the WCET in the following way. Assume that during execution some units $u_1$ and $u_2$ execute between 5 and 20 times each. Using unary hypotheses only, this execution information is turned into the two constraints $5 \leq x_1 \leq 20$ and $5 \leq x_2 \leq 20$. During IPET calculation, the partial assignment $x_1 = 20 \land x_2 = 20$ satisfies the constraints. If however there is a correlation between the execution of the units $u_1$ and $u_2$ because of control or data dependencies, and the correlation is that the constraint $10 \leq x_1 + x_2 \leq 30$ is true for all observed executions, then adding this constraint helps tighten the WCET estimation since the previous (maximum-value) assignment $x_1 = 20 \land x_2 = 20$ is not valid any more.

Row 7 captures execution-count frequency relations. For example, assume that some unit $u_1$ executes with an observed hypothesis $1 \leq x_1 \leq 4$, that some unit $u_2$ executes with an observed hypothesis $4 \leq x_2 \leq 16$, that $u_1$ is one of two alternatives of a conditional statement and has a tiny execution time $c_1$ compared to the other alternative, and that $u_2$ has a large execution time $c_2$. During the IPET calculation, maximizing the execution time could potentially force the IPET solver to choose $x_1 = 1 \land x_2 = 16$ i.e. maximize the number of occurrences of $u_2$ and minimize the number of occurrences of $u_1$ in the longest path (minimizing the number of occurrences of $u_1$ means maximizing the number
of occurrences of the other alternative of the conditional statement where \( u_1 \) resides — which in this example has a larger execution time). However, if a constraint of the form \( x_2 = 4 \times x_1 \) is added to the program model, then if the IPET calculation decides that \( x_2 = 16 \), it will be forced to make \( x_1 = 4 \) which tightens the WCET since unit \( u_1 \) with the tiny execution time executes more along the longest path.

3) Higher-Arity Hypotheses: Hypotheses of higher arity can also be derived. The accuracy of the WCET estimation can potentially improve by using higher-arity hypotheses since they can express more correlations between execution counts. Thus, ternary constraints can potentially bind the execution counts more strongly than binary constraints can, and quaternary constraints are stronger than ternary constraints in expression correlations between execution counts, and so on.

4) \( d \)-Arity Hypotheses: The inclusion of \( d \)-arity hypotheses \((d \geq 1)\) will generally tighten the WCET estimation. The higher the arity \( d \), the more correlation between execution counts is captured. However, capturing more correlation does not necessarily mean that more tightness can be achieved in WCET estimation. For example, the ternary hypothesis \( x_1 + x_2 + x_3 \leq 15 \) is weaker than the conjunction of binary and unary hypotheses \( x_1 + x_2 \leq 10 \wedge x_3 \leq 5 \) since the latter implies the former but not the opposite.

B. Hypothesis Generation

Deriving hypotheses from execution-count matrix \( X \) is straightforward. For unary hypotheses, we traverse the column \( X \) corresponding to the program unit of interest. In order to derive an upper (respectively lower) bound on the execution count of some unit \( u_i \), we find the maximum (respectively minimum) of the column \( X[j][i], j \in [1..m] \). In order to derive an exact execution count for unit \( u_i \), the column \( X[j][i], j \in [1..m] \) is traversed checking if all its cell values are equal to each other. It takes linear time \( \Theta(m) \) to find the maximum, the minimum, or to check if all cells of the column \( X[j][i], j \in [1..m] \) have identical values. There are three unary hypotheses to derive over \( n \) program units, and hence the overall complexity is \( \Theta(3.n.m) \).

For our binary hypotheses, the ordering of the arguments (Table II) is not important and so we shall refer to binary sets of arguments as opposed to pairs of arguments in the following — except the hypothesis of Row 7 which is broken down to two symmetric hypotheses \( x_{i1} = f_{i1,i2} \cdot x_{i2} \) and \( f_{i1,i2} \cdot x_{i1} = x_{i2} \). There are four hypotheses to be derived over \( n(n-1)/2 \) binary sets of program units, and after splitting the hypothesis of Row 7, the number of binary hypotheses becomes five. It takes linear time \( \Theta(m) \) to find the maximum or the minimum of a pairwise sum of two columns from matrix \( X \). It also takes linear time to check if all cells of the columns \( X[j][i_1], j \in [1..m] \) and \( X[j][i_2], j \in [1..m] \) obey one linear relation of the form \( x_{i1} = f_{i1,i2} \cdot x_{i2} \) or \( f_{i1,i2} \cdot x_{i1} = x_{i2} \). Therefore, the complexity is \( \Theta((5/2).n(n-1).m) \). For example, in order to find frequency relations, we would visit all \( n(n-1)/2 \) possible binary sets \( \{i_1, i_2\} \) of columns indexed by \( i_1 \) and \( i_2 \); and for each binary set, we traverse the rows \( j \in [1..m] \) one by one trying to establish a relation \( (X[j][i_1] = f_{i1,i2} \cdot X[j][i_2]) \) or \( (f_{i1,i2} \cdot X[j][i_1] = X[j][i_2]) \) for \( f_{i1,i2} \in \mathbb{N} \).

For ternary hypotheses, our hypothesis derivation has \( O(m.n^3) \) computational complexity, and so on.

C. Input-Data Generation

In order to increase the likelihood of deriving correct hypotheses, input-data generation should give rise to execution traces characterised by adequate diversity and extreme-value execution counts as follows.

First, in order to explain the reason behind requiring diversity in the execution traces let us consider the following scenario. Assume that all execution traces contain \( x_1 = 1 \) and \( x_2 = 2 \) for some units \( u_1 \) and \( u_2 \). In this case, the likelihood of the derived hypotheses being wrong is potentially high since the units \( u_1 \) and \( u_2 \) have possibly not been exercised through different execution paths. This means that the execution counts \( x_1 \) and \( x_2 \) could potentially show a strong and false correlation between the executions of \( u_1 \) and \( u_2 \) in a time when they are not correlated in reality. Hypotheses of arity 2 and above benefit from the diversity in collected values of execution counts per program unit. In terms of the \( m \)-by-\( n \) execution-count matrix \( X \); in order to achieve our goal of increasing the likelihood of deriving correct hypotheses, we need to obtain a matrix \( X \) where each of its columns has got a high standard deviation.

Second, maximizing the execution counts per trace is also a way of increasing the likelihood of deriving correct hypotheses. For example, a hypothesis expressed as an upper bound on some execution count is likely to be correct if we generate the largest possible value for the execution count during testing. In a symmetric way, minimizing the execution counts benefit hypotheses that express lower bounds on execution counts.

We treat the test-data generation problem as an optimization problem and generate input vectors using genetic algorithms.
The idea is to evolve generations of input vectors while trying to maximize their fitness all the way through.

We proceed as follows. Initially, the matrix $X$ of execution counts is empty. We use three kinds of GA-based test-data generation. The first one that we call $GA_d$ is used for filling in the matrix $X$ with execution counts suitable for deriving hypotheses that are likely to be correct if the columns of $X$ have high standard deviations. The second one that we call $GA_u$ is used for filling in the matrix $X$ with execution counts suitable for deriving hypotheses that are likely to be correct if the execution-count values are maximized. The second one that we call $GA_l$ is used for filling in the matrix $X$ with execution counts suitable for deriving hypotheses that are likely to be correct if the execution-count values are minimized.

The reason behind using three different GAs is the following. The test-data generation in our case attempts to increase the likelihood of the correctness of the derived hypotheses in different directions. For example, some hypotheses require diversity in the execution counts per program unit and some require extreme-value execution counts. Trying to maximize or minimize the values of the execution counts lead to a per execution-count distribution which is biased towards maximal or minimal values i.e. characterised by poor diversity. Similarly, maximizing execution counts pushes test-data generation in the opposite direction of minimizing them.

Using a multi-objective GA is another way of combining all the (conflicting) objectives. However, a suitable aggregate objective function and appropriate Pareto-based ranking schemes need to be researched carefully which is outside the scope of this work. For simplicity, we prefer to evolve three populations of input vectors corresponding to the three GAs $GA_d$, $GA_u$, and $GA_l$.

The fitness of an input vector in $GA_u$ or $GA_l$ is the sum of the execution counts along its corresponding row in the matrix $X$. During the evolution of the GA, rows of large execution-count sums are generated (in the case of $GA_u$): the values of the individual execution counts $x_i$ in any given row vary from generation to generation in the GA. In theory, maximizing the sum of execution counts could cause Pareto behaviour i.e. the individual execution counts in a given row are (almost) equal to each other while maximizing their sum. However, witnessing such behaviour in reality has got a tiny probability of occurring given that different program units have different lower and upper bounds on their number of occurrences in the execution trace which potentially renders Pareto optimality impossible, and also because of the crossover and mutation that occurs in the GA which changes the values of the individual execution counts in the row from generation to generation, thus causing the different execution counts to contribute to the sum with different percentages across the generations of the GA. A similar (symmetric) argument applies to the case of $GA_l$.

The fitness of an input vector in $GA_d$ is more complex because it depends on the contribution of the input vector’s corresponding execution counts to the diversity of the execution counts per column in matrix $X$. In other words, we cannot evaluate the fitness of the input vector by looking at its corresponding row in $X$ alone, we have to account for all other rows to reason about diversity. Our means of generating input vectors that enforce diversity is by considering the matrix $X$ as the phenotype of an individual of the population of $GA_d$ which is a set of input vectors (the genotype) and evolve it over the generations. The fitness of an individual, a set of input vectors in this case, is the sum of the standard deviations of the columns of the corresponding matrix $X$. The larger the sum of the standard deviations, the more diversity to be expected. Again, a Pareto behaviour can potentially be observed where the standard deviations are not maximal, yet they lead to a large sum; however, we can conduct a similar argument to what we did before and say that the Pareto behaviour has tiny chances of occurring.

In $GA_d$, a population of sets of input vectors is evolved: each element of the set corresponds to a row in $X$, and the whole set corresponds to $X$ itself. The evaluation of the fitness of each individual (set of vectors) necessitates running the program under analysis for each input vector in the set, for all sets, and during all the generations of the GA. The overall time complexity of running $GA_d$ is $O(g.k.m.n)$ where $g$ is the number of generations and $k$ is the number of individuals in the population; and the space complexity is $O(k.m.n)$. The evolution of $GA_d$ is inherently slow, however, alternative static-analysis solutions to the flow-analysis problem such as abstract interpretation and symbolic execution are known to be extremely slow.

In summary, we use three different test-data generation methods: one for maximizing execution counts, one for minimizing execution counts, and one for maximizing the standard deviation of execution counts of the same program unit. The size of the population in $GA_u$ and $GA_l$ is $m/3$, and the size of each individual in $GA_d$ is $m/3$. The result of running $GA_u$ and $GA_l$ are two matrices $X_u$ and $X_l$ of $m/3$ distinct rows each (the GA is instructed to discard duplicate individuals in any given generation). The result of running $GA_d$ is a population of $k$ matrices of $m/3$ rows each; where the fittest matrix $X_d$ is then selected as the population of input vectors with best standard deviation. The three matrices $X_u$, $X_l$, and $X_d$ are concatenated to form the execution-count matrix $X$ where deriving hypotheses takes place.

D. Hypothesis Validation

Once we generate our hypotheses, we must prove their correctness. In order to do this, we use a bounded model checker which takes as input the program $p$ and the hypotheses as assertions. The model checker either proves the assertions in which case the hypotheses become established flow facts or generate an input vector that violates at least one of the assertions. This input vector is used to execute program $p$ one more time, append an extra row to the matrix $X$, re-perform the hypothesis derivation, and generates hypotheses that do not fail the previous counter example.

This cyclic process of deriving hypotheses from matrix $X$, validating them using model checking, executing program
Fig. 1. The steps of deriving flow-fact hypotheses and validating them to become flow-fact constraints added to the IPET model of the program.

with counter-example input vectors, updating \( X \) is repeated a number of times until either the model checker validates all hypotheses or a time-out on the analysis time is reached. Figure 1 shows in more detail the steps of our analysis.

### IV. Evaluation

We will evaluate our approach with respect to the following criteria. First, we evaluate the tightness introduced in the WCET estimation by adding the flow-fact constraints. Second, we evaluate the goodness of our proposed test-data generation in terms of speeding-up model checking by increasing the likelihood of deriving correct flow-fact hypotheses.

#### A. Evaluation Framework

The evaluation is performed according to the framework depicted in Figure 2 which starts with a C program and finishes by producing a WCET estimation for it. The C program input to our analysis comes from the Malardalen benchmarks [15] which are traditionally used for evaluating WCET-analysis results.

The first stage of the evaluation is test-data generation and building the execution-count matrix.

Let us first discuss the experimental setup for capturing the execution counts. In order to obtain the execution counts of the different program units during execution, we use the simplescalar cycle-accurate simulator [4] to produce a set of traces which can be parsed to identify the number of occurrences of each program unit along an execution path (a single trace). Notice that the central idea of this work is estimating the WCET using observed and proved flow facts derived from traces i.e. the source of the traces is orthogonal.

The same effect can be obtained using a hardware debug interface which uniquely identifies executing program units. In a later stage, the execution-count information derived by executing the binary code must be transformed into assertions at the source code i.e. some form of mapping execution information is needed. This mapping is necessary because the model checker we are using works at the source-code level; if the model checker can work at the object-code level then no mapping is needed. In our case, throughout the course of working with the Malardalen benchmarks, we have created mapping files for various programs to link execution information from object code to source code which we shall use here; in other words the mapping is done using manually-created maps. In the general case, execution-information mapping techniques can be used [12], or the source-code is instrumented and the execution counts are recorded for the instrumentation points; which is more elegant but suffers the instrumentation overhead.

Let us then discuss test-data generation. In order to show the goodness of our test-data generation technique, we compare it to two other test-data generation methods. Figure 2 depicts three test suites: test suite 1 for test-data generation for hypothesis derivation like discussed in Section III-C, test suite 2 for test-data generation using GAs to maximize execution times, and test suite 3 for test-data generation using random selection of input vectors. The rationale is the following. Using test suite 2, we want to see if simply generating input vectors for maximizing execution times yield execution-count matrices suitable for deriving hypotheses with high likelihood of being correct. Notice that the GA \( G_A \) used in test suite 2 is different from the GA \( G_A \) (Section III-C) since the former maximizes execution times and the latter maximizes execution counts. The reason behind using test suite 3 is to see if a much-cheaper test-data generation such as random testing can beat the relatively-expensive test-data generation used in test suite 1. Test suites 1 and 2 both need feedback from trace parsing which is not shown in Figure III-C for simplicity.

For all programs used in evaluation, the GAs \( G_A \), \( G_A \), \( G_A \), and \( G_A \) all operate on a population of 100 individuals evolved over 100 generations. The size of the individual in \( G_A \) and \( G_A \) is the number \( n \) of program units \( u_i \). The size of the individual in \( G_A \) is \((1/3)m\) \( n \) which is the size of a third of the execution-count matrix \( X \) (the other two thirds are reserved for the results of \( G_A \) and \( G_A \)). The size of the individual in \( G_A \) is the size \( |v| \) of the input vector \( v \) to the program being analysed. The number of rows of the execution-count matrix is \( m = 300 \) i.e. at most 300 flow facts can be derived for any program. These values are based on engineering wisdom and computational-effort trade-off analysis.

For test suite 3, we use the timing model obtained by \( G_A \). The reason for this, is that the timing model obtained by random testing is likely to not force maximum execution times of program units. Therefore, in our random testing, we use a timing model supplied by a third party analysis, and focus on the derivation of hypotheses from an execution-count matrix derived by random test-data generation. We took this measure
to avoid that the WCET estimation using random test-data
generation becomes unsafe because of poor execution-time
coverage.

The second stage of the evaluation is hypothesis derivation
which corresponds to the flowchart of Figure 1. We use the
CBMC model checker [6] in order to prove the correctness of
hypotheses or falsify them by generating counter examples.

The third and final stage of the analysis is calculating the
WCET by solving the IPET model containing the structural
constraints that express flow preservation, the time constraints
which are the maximum execution times scored for every unit
during testing, and flow-fact constraints which are the result
of hypothesis derivation. The IPET model is ILP-based and so
we use lp_solve [2] to solve it.

We have applied the framework of Figure 2 on a subset
of the benchmarks in [15]. We have modified the set of
benchmark programs in order to make them amenable to
a measurement-based analysis (MBA) framework. 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.

We have used a relatively complex hardware configuration
for the cycle-accurate simulator with out-of-order pipeline,
instruction cache, and dynamic branch prediction. The reason
for choosing a complex hardware configuration is twofold.
First, we want the hardware configuration to be representative
of the kind of hardware used in embedded systems nowa-
days. Second, the test-data generation based on the GA that
maximizes execution times (test suite 2) reduces to test-data
generation with the GA that maximizes execution counts ($GA_u$
in test suite 1) in case of using simple hardware; because
in the latter, execution-time variability is small and hence
maximizing execution times is achieved mostly by maximizing
execution counts as opposed to also maximizing cache misses
for example.

B. Results

1) Tightness: From own previous work [3], [16], we have
collected the WCETs of the benchmarks of Table III through
very-expensive exhaustive path testing, and we list them in
the second column labelled $W$. These values will help us
determine the tightness of our WCET estimations. The column
labelled $W_1$ contains the WCET estimation based on the timing
model derived during testing and the validated hypotheses
derived from the execution-count matrix corresponding to test
suite 1. The column labelled $P_1$ contains the percentage of
pessimism between the WCET estimations in column $W_1$ and
the real WCET values $W$. Similarly, column $W_2$ contains
the WCET estimations enhanced by flow facts generated
from the execution-count matrix built from testing suite 2,
and column $P_2$ shows the corresponding pessimism. Finally,
column $W_3$ contains the WCET estimations enhanced by flow facts generated from the execution-count matrix built
from testing suite 3, and column $P_3$ shows the corresponding
pessimism.

Table III shows that for all programs, the tightest WCET es-

timation is the one resulting from the IPET model enhanced by
the flow-fact constraints resulting from the test-data generation
that enforces maximum-value execution counts, minimum-
value execution counts, and execution-count distributions with
high standard deviations. The next tighter WCET estimation is
the one resulting from solving the IPET model enhanced with
flow facts obtained from test-data generation that maximizes
execution times. The least-tight WCET estimation is the one
obtained by solving an IPET model augmented with flow
constraints obtained by using random test-data generation.

The program timing models that we obtained from test suite
1 and test suite 2 are almost identical. This means that the
tightness in the WCET estimations in column $W_1$ is indeed
due to the quality of the derived flow facts as opposed to
less-pessimistic timing model. In fact, during experiments, the
derived flow facts — for each program of Table III — from
test suite 2 were always a subset of the derived flow facts
using test suite 1. The extra flow facts derived by using test
suite 1 are responsible for the extra tightness. The generated
flow facts using test suite 3 are also a subset of the generated
flow facts using test suite 2. We would expect this statistic to
potentially vary when the experiment is repeated because of
the “random” test-data generation involved.

2) Scalability: The approach presented in this paper is
expensive, it uses a GA and a model checker. Nevertheless, it
provides correct flow facts. Notice that alternative approaches
such as using abstract interpretation to perform flow anal-

ysis are also expensive — and this is why the Malardalen
benchmarks are tailored to be input independent to reduce the
number of execution paths significantly to make flow-analysis
based on abstract interpretation possible (in our evaluation, we
modified them to make them more data dependent). Certainly


alternatives are also expensive — and this is why the Malardalen
benchmarks are tailored to be input independent to reduce the
number of execution paths significantly to make flow-analysis
based on abstract interpretation possible (in our evaluation, we
modified them to make them more data dependent). Certainly


tics such as abstract interpretation are faster than model
checking because the former cuts on the number of states by
using abstraction — at the cost of losing precision — while
the latter uses the actual concrete states.

The GA’s execution time depends on the execution time of
the program being analysed because the GA needs to execute
the program for different input vectors to obtain their fitness by
collecting the execution counts of their units. The runtime of
the GA (including $GA_u$, $GA_l$, $GA_d$, and $GA_t$) never exceeded
one hour for any given program.

The model checking of the flow facts generated by test
suite 1 completed successfully for 3/5 of the benchmarks
and timed-out for at least one hypothesis for the remaining
2/5 benchmarks. For each program, all the flow facts were
either proved correct or the model checking timed-out. The
analysis never ran into a situation where the model checker
produces a counter example that forces the analysis to re-
derive hypotheses. This does not prove that none of the
hypotheses is false, but it shows that none of them is easy
to prove false. The model checker was left running a total
time of 17 hours.
The model checking of the flow facts generated by test suite 2 and test suite 3 completed successfully for 1/3 of the benchmarks and timed-out for at least one hypothesis for the remaining 2/3 benchmarks. The model checker generated numerous counter examples that forced hypothesis re-derivation. The model checking (including re-derivation) ran for 28 hours. This shows that analysis time can be enhanced by using a suitable test-data generation method optimised for deriving hypotheses with high likelihood of being correct.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have shown a fully-automatic method that derives exact flow facts for WCET analysis based on careful test-data generating and model checking. The tightness of WCET estimates obtained using our method demonstrates the usefulness of our approach.

We intend to extend this work to more industrial-strength programs and also compare the flow facts derived by our analysis with those derived by a static-analysis tool subject to availability.

### TABLE III

<table>
<thead>
<tr>
<th>Program</th>
<th>$W$</th>
<th>$W_1$</th>
<th>$P_1$</th>
<th>$W_2$</th>
<th>$P_2$</th>
<th>$W_3$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary_search</td>
<td>8554</td>
<td>8582</td>
<td>0.0%</td>
<td>8752</td>
<td>2%</td>
<td>8767</td>
<td>2%</td>
</tr>
<tr>
<td>bubblesort</td>
<td>37066</td>
<td>37066</td>
<td>0%</td>
<td>37066</td>
<td>0%</td>
<td>37066</td>
<td>0%</td>
</tr>
<tr>
<td>crc</td>
<td>1593775</td>
<td>1593775</td>
<td>0%</td>
<td>1593775</td>
<td>0%</td>
<td>1593775</td>
<td>0%</td>
</tr>
<tr>
<td>dct</td>
<td>499942</td>
<td>499950</td>
<td>0.0%</td>
<td>499954</td>
<td>0.0%</td>
<td>499954</td>
<td>0.0%</td>
</tr>
<tr>
<td>expint</td>
<td>6200</td>
<td>6346</td>
<td>1%</td>
<td>6442</td>
<td>3%</td>
<td>7333</td>
<td>17%</td>
</tr>
<tr>
<td>factorial</td>
<td>7556</td>
<td>7938</td>
<td>5%</td>
<td>8123</td>
<td>8%</td>
<td>9974</td>
<td>32%</td>
</tr>
<tr>
<td>fdct</td>
<td>503200</td>
<td>503200</td>
<td>0%</td>
<td>503200</td>
<td>0%</td>
<td>503200</td>
<td>0%</td>
</tr>
<tr>
<td>fbcall</td>
<td>58704</td>
<td>58790</td>
<td>0.0%</td>
<td>58790</td>
<td>0.0%</td>
<td>60112</td>
<td>2%</td>
</tr>
<tr>
<td>fir</td>
<td>5231162</td>
<td>5603004</td>
<td>7%</td>
<td>5982045</td>
<td>14%</td>
<td>6406842</td>
<td>22%</td>
</tr>
<tr>
<td>insertsort</td>
<td>32371</td>
<td>32426</td>
<td>0.0%</td>
<td>32514</td>
<td>0.0%</td>
<td>32514</td>
<td>0.0%</td>
</tr>
<tr>
<td>janne_complex</td>
<td>19091</td>
<td>19391</td>
<td>2%</td>
<td>20674</td>
<td>8%</td>
<td>24573</td>
<td>29%</td>
</tr>
<tr>
<td>matrix_cint</td>
<td>138754</td>
<td>142163</td>
<td>2%</td>
<td>154945</td>
<td>12%</td>
<td>194451</td>
<td>40%</td>
</tr>
<tr>
<td>matrix_mult</td>
<td>893545</td>
<td>926204</td>
<td>4%</td>
<td>988055</td>
<td>10%</td>
<td>1109888</td>
<td>24%</td>
</tr>
<tr>
<td>select</td>
<td>94011</td>
<td>98420</td>
<td>3%</td>
<td>112466</td>
<td>20%</td>
<td>149638</td>
<td>59%</td>
</tr>
</tbody>
</table>

In this paper we have shown a fully-automatic method that derives exact flow facts for WCET analysis based on careful test-data generating and model checking. The tightness of WCET estimates obtained using our method demonstrates the usefulness of our approach.

We intend to extend this work to more industrial-strength programs and also compare the flow facts derived by our analysis with those derived by a static-analysis tool subject to availability.

207
REFERENCES


