UNIVERSITY of York

This is a repository copy of *TACO: : An industrial case study of Test Automation for COverage*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/137444/

Version: Accepted Version

Proceedings Paper:

Lesage, Benjamin Michael Jean-Rene, Law, Stephen Andrew and Bate, Iain John orcid.org/0000-0003-2415-8219 (2018) TACO: : An industrial case study of Test Automation for COverage. In: Proceedings of the 26th International Conference on Real-Time Networks and Systems. 26th International Conference on Real-Time Networks and Systems, 10-12 Oct 2018, LIAS/ISAE-ENSMA. RTNS'18 . ACM , pp. 114-124.

https://doi.org/10.1145/3273905.3273910

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

TACO: An industrial case study of Test Automation for COverage

Benjamin Lesage
University of York
United-Kingdom
name.surname@york.ac.uk

Stephen Law
Rolls-Royce Controls Systems
United-Kingdom
name.surname@controlsdata.com

Iain Bate
University of York
United-Kingdom
name.surname@york.ac.uk

ABSTRACT

Timing analysis is an important part of the development of critical real-time systems. It stems from the need to provide evidence on the behaviour of the system, compliance to requirements and timing bounds. The formal testing process is complicated, and includes tests to achieve compliance with certification requirements. Where possible, testing should be performed on a host and then validated on the target. This is especially important for real systems where the target may not be available early in the project or target-based testing is expensive and time consuming. Meaningful host-based testing is difficult when it comes to timing analysis. Automation helps reduce the costs and move testing earlier in the application development cycle. Moving testing earlier in the development cycle not only enables the testing to scale to whole systems, it allows the risks of projects to be managed and software to be optimised before target-based testing is performed.

In this paper, we extend existing work achieving reliable coverage and High WaterMark (HWM) measurement, to scale its application to the analysis of a full system software build, automate the test process, and minimise the set of tests deployed on target. Our case study demonstrates the successful application of the approach on a large code base, i.e. an existing controls system software code. The paper ends with a position statement about how this work is instrumental for both future research but also as part of industry practically analysing the timing behaviour of systems automatically and certifying mixed-criticality systems.

ACM Reference Format:
Benjamin Lesage, Stephen Law, and Iain Bate. 2018. TACO: An industrial case study of Test Automation for COverage. In *26th International Conference on Real-Time Networks and Systems (RTNS '18), October 10–12, 2018, Chasseneuil-du-Poitou, France.* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/3273905.3273910

1 INTRODUCTION

Testing is an important part of any engineering process and safety-critical software engineering is no exception. One area of software testing important to many safety-critical systems is to support timing analysis as these systems tend to be hard real-time systems, i.e where a failure to meet the system's timing requirements is potentially catastrophic. As the focus of this paper is timing, the terms tests and testing are not using the strict DO-178C definitions which refer to verifying that functional requirements are met. Instead, *testing* refers to executing the software under a set of *test* conditions, where an individual test includes the inputs to a code item and the configuration of the system, in order to understand the timing behaviour of a system.

In this paper, our focus is obtaining execution time traces on an industrial scale as part of Measurement-Based Timing Analysis (MBTA). The context for the work is in the design of Full Authority Digital Engine Controllers (FADEC) that are designed according to DO-178C DAL A guidelines for certification [17]. Our aims are to reliably find execution time bounds for a task, either as a HWM close to the actual Worst-Case Execution Time (WCET) or supporting WCET analysis with tools such as RapiTime [16]. The requirement in particular is to obtain full coverage of the executed code to provide confidence in an efficient and reliable fashion that de-risks the FADEC development early in the design life-cycle. As full path coverage is not possible, a realistically achievable level of coverage is aimed for. The work is also a step towards the analysis of DAL C software and, as part of this, the framework has been tested with a more advanced processor platform, i.e. a Raspberry PI3.

The work is also a step towards the analysis of DAL C systems as part of a wider study on the potential of Mixed-Criticality Systems (MCS). The timing estimates C_i for a task can be derived by analyses dictated by its criticality level i with higher criticality levels relying on more conservative analysis, e.g. using the observed HWM as C_C or C_{Lo} and predicted WCET through RapiTime as C_A or C_{Hi}.

In [14], Law proposed a search-based algorithm that could reliably provide better coverage data and WCET estimates than existing state of the art search algorithms when applied to existing benchmarks [8], as well as some of the FADEC software on Rolls-Royce's target flight hardware. In this paper, *reliably* means it normally gets more coverage as well as a larger HWM and estimated WCET than the other approaches given a finite amount of time, and *better* means the HWM is closest to that ever observed across a very large amount of tests. The results were shown to be both statistically and scientifically significant. Reliable is important as one off good results are insufficient for both designing the FADEC but more importantly as part of justifying the process during certification.

One of the proposed fitness function, *BCHLr*, in particular reliably gets the best executed code coverage, HWM and WCET estimates. *BCHLr* combined multiple metrics: a form of block coverage that not only counted how many blocks were executed but also considered local path coverage, i.e. whether all feasible routes (termed edges) exiting the blocks have been covered; a form of loop counts; and finally reverts to a previous solution if progress wasn't being made. The important driver for these metrics is they are based on structural coverage, which DO178-C uses to assess

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RTNS '18, October 10–12, 2018, Chasseneuil-du-Poitou, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6463-8/18/10…$15.00
https://doi.org/10.1145/3273905.3273910

the scope of functional tests, meaning the fitness function does not require execution times to generate the *best* test vectors.

The contributions of this paper are as follows:

- Produce an automated test framework - Test Automation for COverage (TACO), defining analyses processes based on coverage and timing data.
- Scaling the approach in [14] to a large FADEC system.
- Support extensive tests on low-cost platforms, e.g. Raspberry PI3 (PI3) and the host desktop PCs.
- Demonstrate that tests on the host correlate with the final results on the target in terms of coverage.

The structure of the paper is as follows. Section 2 discusses related work especially in the domain of measurement-based timing analysis and test case generation. We summarise the contents of [14] that are relevant for understanding the fitness function that drives the search-based algorithm in our framework in Section 4. Section 5 provides an overview of the framework, and Section 6 discusses its application to different target platforms. We present the results of the application of the TACO framework to a FADEC in Section 7. The results of our case study led to the position statement in Section 8 as part of the conclusions.

2 RELATED WORK

This section presents a study of the existing work in the field of Measurement-Based Timing Analysis (MBTA) and Automatic Test Case Generation (ATCG).

2.1 Measurement-Based Time Analysis

MBTA techniques rely on measuring the code items under analysis on target to derive their Worst-Case Execution Time (WCET). The provision of a sound bound however requires consideration of an oft intractable input space, combining numerous factors, e.g. input variables, system and platform state [22]. A common approach to MBTA is to simplify the problem. This includes reducing the size of the problem, e.g. the number of paths to explore, to more amenable chunks, or extrapolating from a sample of measurements. Deverge & Puaut [5] segment a program and only explore the full paths' set within each segment to avoid the need to exercise the code item under all feasible paths. In a similar vein Stattelmann & Martin [18] present an MBTA tool that also breaks the test code item into a number of easily traceable segments; the WCET is then expressed as an equation of the timing observations across each segment. Both approaches are not practical for large scale industrial projects, as the number of analysed functions and code segments increase with the complexity of the system. So the processing, or engineering, effort required would remain significant. For example a typical electronic engine control system produced by Rolls-Royce contains around 5000 separate functions.

The RapiTime tool from Rapita Systems Ltd is a commercially available WCET analysis tool aimed at the industrial market. The tool breaks the code items down through the use of source code instrumentations, in contrast to [18] however the analysis is performed at the block level, and depending on the target hardware, does not require path coverage.

Measurement-Based Probabilistic Timing Analysis was first proposed by Stewart & Burns [6]. This was later extended by Hansen et al [9] and Cucu-Grosjean et al [4]. Provided the observations fed into MBPTA follow some strict statistical properties, MBPTA can produce statistical upper-bounds on the temporal behaviour of a code item using Extreme Value Theory (EVT). A recent paper by the EVT community highlighted a number of challenges that exist including generating the appropriate test vectors [7].

Ultimately all measurement techniques are dependent on the quality of the data fed into the analysis. The nature of this data is analysis dependent but the quality of this data, and hence the quality of the analysis result, is a direct result of the inputs exercised on the item under analysis and the testing environment. The previous work in the field has focused on generating results with a given known good data set, or in the case of [4] using hardware randomisation to force hardware to make a data set *good*.

2.2 Automatic Test Case Generation

Wegener [20] and Tracey [19] both illustrated how search algorithms could be used for test data generation, particularly with regard to applications which requirements go beyond simple statement coverage, e.g. Modified Condition/Decision Coverage requirements for adequate testing [17].

Wegener's early work [20] built off Jones et al [11] and presented an investigation into how genetic algorithms can be used to estimate the minimum and maximum execution times of software targeting embedded systems. Tracey introduced a framework of tools designed to automatically generate test data to perform dynamic analysis on an test code item. One of the targeted analyses being the analysis of the WCET. The work was targeted towards safety-critical systems using strongly typed Ada [19]. The framework introduced is primarily based on search algorithms, which when compared to system HWM observations, produced good results. However the drawback was that the tool had to achieve path coverage to obtain a sound WCET.

Wenzel [21] introduces an MBTA tool designed to calculate safe WCET bounds of safety-critical software. This is achieved through a combination of test data reuse, random search, genetic algorithms and finally model checking [21]. Unfortunately the tool places a number of restrictions, and assumptions on the code under test, for example the tool is only capable of analysing acyclic code and does not allow function calls.

Williams [23] proposes a static analysis tool which aims to identify a test vector to exercise every path through the code under test. The WCET can then be read off as the HWM observed during testing. This was extended in [24] with an analysis into possible simplifications that can be made to avoid the analysis requiring full path coverage, this includes maximising loop counts, and assuming branches are always taken. The paper recognises that further investigation and justification is required, but it does indicate possible areas where MBTA coverage requirements could be simplified.

Bünte et al [3] examined the effectiveness of using model checking [10] to produce test suites with enough coverage to provide reliable WCET estimates. Their research focuses on identifying effective coverage metrics to drive a model checking test suite generator. This was extended in [2] which combines the results produced with a genetic algorithm, which then aims to identify larger execution times. Drawbacks include the software being analysed

has to be simplified to ensure each decision point relies on only a single variable and the tool's use of model checking risks the tool's portability to larger, more complex functionality.

Khan and Bate [1, 12] introduce the idea of incorporating multi-criteria optimisations into a search based WCET analysis tool using advanced processor features known to cause larger WCET values, such as cache misses, and also focused in on low level software coverage such as loop iterations. The paper concluded that no one fitness function provided better results across all test code items, and that the fitness function chosen should be dependant on the target environment.

In our previous work [14], a different approach to search-based WCET analysis is taken. Instead of focussing on timing, the focus is on path coverage but due to the size of real software a pragmatic approach is taken based on block coverage, local path history and loop counts. More details can be found in Section 4. The approach was shown to reliably provide a higher block coverage, observed and analysed execution times than other fitness functions including maximising the HWM given a finite amount of tests. Due to reasons of space, only a limited number of benchmarks [8] and FADEC tasks were presented. The results were based on a cycle accurate simulator of a deterministic bespoke avionics processor designed by Rolls-Royce.

3 INDUSTRIAL CONTEXT

The work builds upon the current industrial setup deployed within Rolls-Royce, as described by [15]. The FADEC software and processing architectures are amenable to analysis. The control software uses standardized interfaces between the various application layers and the hardware, with well-defined parameters and value ranges. The code uses robustly defined types. In particular, the types of inputs are known, bounded, and verified against requirements. The Rolls-Royce platform, described in Section 3.1, provides accurate tracing and timestamping during code execution. Facilities exist such that tests can be queued, executed and their results collated. The executed code of test items (individual functions) is extensively covered by low-level tests. The code under test and the code in the final system are compared to prevent any discrepancies.

The coverage provided by low-level tests allows for the collection of timings across all blocks of all test items, where each block is a sequence of straight code without outgoing or incoming branches. The individual blocks' timing are then combined by RapiTime [16] to compute the Worst-Case Execution Path and associated WCET.

3.1 The Rolls-Royce Platform

The Rolls-Royce Processor, the VISIUMCORE, is a packaged device that integrates a core, memory, IO and tracing interfaces. Being targeted at the safety-critical embedded sector, the device is DO-254 Level A compliant. It has extensive single-event-upset protection and is suitable for harsh environments. The processor does not incorporate a data or instruction cache. The processor is the target processor for all Rolls-Royce DAL A control system applications, and has been in use and in service since 2017.

The processor has been carefully designed to ensure that each instruction's execution is time-invariant; in other words each instruction will take the same time to execute, regardless of the data its operation is performed upon. These design features ensure that previous processor state has no effect on the current operation of the device; the execution and timing of a block of code is not impacted by prior execution path. There is thus a strong correlation between coverage of an analysed test item blocks and *better* computed timing estimates through MBTA tools.

To enable timing of functions, the processor provides facilities to non-intrusively collect an entire instruction trace complete with timestamps. The processor has also been augmented with functionality to output a user-specified value and timestamp. Both the trace facility and the instruction are low-overhead, incurring only a single instruction fetch.

4 COVERAGE TECHNIQUE [14]

The investigation in [14] identifies how effectively a basic search algorithm could be at generating data for a hybrid MBTA tool, in this case RapiTime [16]. The algorithm supplants low-level tests in driving executed code for coverage. Where a low-level test executes a sequence of manually defined test vectors, the algorithm starts from a random test vector and defines the next test vector to evaluate based on the current test vector, past observations, and its internal state.

We focus on one fitness function, BCHLr. BCHLr was the most reliable one in terms of coverage, HWM, and analysed WCET from [14] and has the additional benefit of not using execution times as these clearly change between targets. We further rely on Ran, a wholly pseudo-random fitness function, to provide a baseline for experimentations.

Example: *In the following, we consider the program in Algorithm 1 to illustrate the behaviour of the algorithm and the framework. It consists of two code items,* `f` *and* `g`*, each including multiple paths selected according to the values of different variables, respectively* `A` *and* `D`*. The types of all variables are constrained and their value interval is known.* `B` *and* `C` *are inputs of function* `f`*, while* `A` *is part of the system state. A minimal test for exhaustive code coverage of* `f` *would go through input vectors:* `{A: 1.0, B: True, C: 4}` *(B1, B2, B3, B5, B6, B8, B7), and* `{A: 1.0, B:False, C: 0}` *(B1, B2, B4, B7).*

4.1 Algorithm Objectives

As the search algorithm executes on the target, coverage and timing measurements are taken across the test code item. Upon completion this entire set of timing measurements are imported into the RapiTime tool. Therefore the aim of the search algorithm is not to execute the worst case path, and identify the WCET. Instead the aim is to obtain high code coverage across the analysed item to ensure the analysed WCET (RWCET), approaches the Actual WCET (AWCET). Note except for the simplest of cases, AWCET is not computable. The technique was designed with specific objectives in mind.

(1) Efficiency - The first objective of the algorithm is to produce results in a reasonable time frame, allowing the analysis to be performed efficiently over a large number of functions. This objective is important as an industrial scale project will be expected to complete a large number of analyses in a restricted time frame, on a limited hardware set.

Algorithm 1 Example of code items

```
1  type Prob is float   range 0..1.0;
2  type Byte is integer range 1..8;
3
4  static A : Prob;
5
6  function f (B: in Boolean, C: Byte)
7    /* B1 */
8    if A > 0.75
9      /* B2 */
10     if B
11       /* B3 */
12       g(C)
13     else
14       /* B4 */
15   /* B7 */
16
17 function g (D: Byte)
18   /* B5 */
19   I : integer range 0..7;
20   I := 0
21   while I < D
22     /* B6 */
23     I := I + 1
24   /* B8 */
```

(2) Consistently the highest instrumentation point coverage - If the test has not achieved good instrumentation point coverage, then the result cannot be trusted as sound. This is because the analysis would have no concept of untested blocks, which could have an effect on the computed RWCET. The objective is assessed overall by capturing the number of portion of instrumented blocks' covered for each code item.
(3) Consistently large execution times, whether HWM or RWCET. This is the ultimate aim of the algorithm, to produce the largest possible computed RWCET result.

4.2 Search Algorithm Setup

The search algorithm used for the analysis is a derivative of the simulated annealing algorithm, originally presented in [13]. The basic algorithm is shown in Algorithm 2. Additional steps, such has reheating [13], are taken to prevent the algorithm being caught in a local minima.

The algorithm starts from a random solution, i.e. a valid test vector for the function under test. On each iteration the `Mutate` function generates a new solution by modifying one randomly selected input in the current test vector. Both operations, generation and mutation, respect the constraints on the function inputs. The analysed code item is then executed using the new solution, and `EvaluateFitness` is used to assess the solution's fitness according to its execution and previous solutions. The new solution is accepted, by the if statement on line 8, if an improvement, or pseudo-randomly selected if a degradation. As the test progresses the pseudo-random selection of worse solutions will decrease, as controlled by `temperature`. Finally `StopAlgorithm` will end the search after no solutions have been accepted for a few iterations (with a basic minimum of 1000 iterations).

Example: *Function* `f` *in Algorithm 1 has three input variables:* `A`, `B`, *and* `C`. *According to the constraints on their type ranges,* `{A: 0.75, B: True, C: 6}` *is a valid random initial solution. The* `Mutate` *operation may opt to randomly mutate* `A`, `{A: 0.8, B: False, C: 6}`, *and then* `B` *in the subsequent call,* `{A: 0.8, B: True, C: 6}`. *The input vector* `{A: 0.0, B: False, C: 12}` *violates the constraints on the values of* `C` *and such solutions cannot be produced by the search algorithm.*

Algorithm 2 Simulated Annealing

```
1  temperature  := InitTemperature()
2  currentSolution := RandomSolution()
3  do
4    newSolution := Mutate(currentSolution)
5    newStats := CallFunction(newSolution)
6    newFitness := EvaluateFitness(newStats)
7
8    if random(0..1) < exp(newFitness / temperature)
9      currentSolution = newSolution
10
11   temperature = DecreaseTemperature(temperature)
12 loop while not StopAlgorithm()
```

4.3 BCHLr Fitness Function

The BCHLr fitness function evaluates the fitness of a new solution, to be accepted or rejected, based on three components:

- *Branch Coverage (BC)* - Accept solutions which cover new branches to execute all branches through the code.
- *Branch History (H)* - Revert to a previous solution that reaches unexplored paths to execute all branches through the code.
- *Maximum Loop Counts (Lr)* - Accept solutions which improve on the observed loop iterations to maximise the iterations of each loop through the code, as proposed by Khan [1].

4.3.1 *Branch Coverage (BC).*

computes the average fitness of the branches traversed through the execution path of the new solution. A branch's fitness is based on the number of edges out of the branch and the number of unexplored edges amongst those. A solution that reaches a branch with yet unseen outgoing edges will be favoured over one that only covers explored branches. The fitness computed through BC depends on the history of explored solutions.

Example: *S* =`{A: 1.0, B: False, C: 1}` *has a lower fitness as an initial solution to the analysis of* `f` *than* `{A: 1.0, B: True, C: 1}` *in Algorithm 1. S covers only 2 unexplored edges, B1 to B2 and B2 to B4, against 3 for the other solution. However, it is the converse if B5 to B6 has already been covered as S covers the unexplored edge B2 to B4.*

4.3.2 *Branch History (H).*

resets the search to previous solutions of interest, that is solutions reaching branches with unseen outgoing edges. As each branch through a solution's path is analysed, the input vector used to drive the current solution is stored against that branch. If a sufficient number of new solutions have been rejected then the set of unseen edges is examined, and the matching branch is chosen at random. The input vector stored against this branch is then adopted as the new input vector. This is designed to attempt to lift the algorithm from poor solutions and focus the algorithm on the area around branches that have only been partially executed.

Example: *Consider B2 has been reached once with solution S =*`{A: 1.0, B: True, C: 7}` *in Algorithm 1, but that the transition from B2 to B4 is un-explored. If the search rejects a sufficient number of solution, as no new edge is explored, currentSolution will revert to*

S before the generation of a new solution to attempt to reach the unexplored edge.

4.3.3 Maximum Loop Counts (Lr). calculates the average number of iterations of each loop on the path traversed by a solution, the result is then normalised using the maximum observed number of iterations. The algorithm is based on previous work by Khan [1].

Example: *Consider function* `g` *in Algorithm 1 with a single input* `D`*. The loops goes through i iterations based on the value of* `D`*. The solution* `{D: 8}` *maximises the number of iteration and thus, in the absence of other branching path, would always be accepted. Conversely,* `{D: 4}` *would be accepted only if no higher value of* `D` *has been considered.*

4.3.4 Combining fitness components. The BCHLr function combines the result produced using BC, with the result using Lr to produce a fitness function that begins by trying to identify unseen blocks, but evolves as the search progresses to favour higher loop counts. Both fitnesses are combined using a weighted sum, with weights W_{Lr} and W_{BC} for Lr and BC respectively:

$$Fitness_{BCHLr} = W_{Lr} \times Fitness_{Lr} + W_{BC} \times Fitness_{BC} \quad (1)$$

As the test progresses, and the branch coverage obtained increases, then the loop fitness Lr weighting (W_{Lr}), is increased (and W_{BC} accordingly decreases). This changes the priority of the fitness function as the test progresses to focus on maximising loop counts.

5 TACO FRAMEWORK

TACO is a framework for the automated generation of test cases. It provides a set of tools to drive any code item by executing test-vectors prepared beforehand or generated on the fly, e.g. using a search algorithm. A driver, such as the simulated annealing algorithm presented in Section 4, can then control the execution of any test code item by selecting the exercised inputs for each execution. The traces resulting from the execution of the code item, both on target or on host platforms, provide information about the set of covered instrumentation points, the executed test vectors, and blocks' timing when available on platform. The resulting information and tools can thus be used to build different analysis processes. As an example, executed code coverage can be obtained on a full system scale by first driving the search through the system entry points, then repeating the process on sub-functions that have not been sufficiently covered.

The application of the TACO framework to a specific test code item, illustrated in Figure 1 and detailed in the subsequent sections, relies on the following steps:

(1) Generating a testport, a function that can execute the code item using a specified test vector;
(2) Instrumenting the code, capture the structure of the code and insert primitives to gather timing and coverage;
(3) Preparing a driver, compile the testport with a test vector driver;
(4) Executing the testport, execute the code item with the selected or generated test vectors and gather data on the execution;
(5) Processing execution traces, identify covered sub-functions, blocks, and their timings.

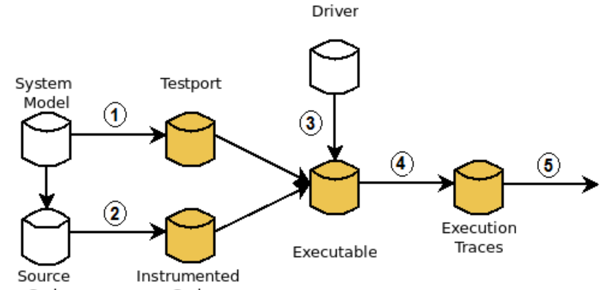

Figure 1: Steps of the TACO Framework

5.1 Generating a testport

The framework relies on a generic interface between driver and executed test item, that is a generic definition of a test vector, such that either can be exchanged as needed. A testport thus provides a single entry point to initialise, run, measure the behaviour of a test item, and provide feedback to the driver on the executed path, from a valid test vector.

The generation of a testport relies on a model of the test item built from its defined requirements and Ada SPARK type annotations. The test vector definition thus captures the number, types, and ranges of all relevant inputs to the test item, including both direct arguments but also relevant parts of the system state. Vectors and structures are expanded down to individual scalars, integer, floating point or boolean, each represented as a single entry in the test vector. As an example, an array of three integers will use three separate integers in the test vector.

The test driver is independent from the test item. While the test driver defines the values in a test vector, it has no direct access to the matching variable in the system build; the testport is responsible for the initialisation of the variables in the system from the driver-provided test vector.

Example: *We use function* `f` *of Algorithm 1 to illustrate the testport generation process. The system model includes all inputs of the function, their types and value range:*

- `A`: *{**in** internal, **float**, **min**: 0, **max**: 1.0}*
- `B`: *{**in** parameter, **integer**, **min**: 1, **max**: 8}*
- `C`: *{**in** parameter, **Boolean**}*

The testport needs to setup the value of internal state variable `A` *based on the input vector and call function* `f` *with the selected values of* `B` *and* `C`*, conceptually:*

```
1 function CallFunction(test)
2   A := test.input['A']
3   StartTracing()
4   f(test.input['B'], test.input['C'])
5   StopTracing()
6   return CollectStats()
```

5.2 Instrumenting the code

Each code item is instrumented by the Rapita Verification Suite, which includes RapiTime [16]. Instrumentation Points are inserted throughout the source code in order to record the program flow, this includes at the start and end of each function, and around conditional statements. Instrumentation points uniquely identify basic blocks in the code. RapiTime further extracts the structure

of the code to generate the structures used by the BCHLr heuristic (Section 4) to track executed edges and loops. In the example of Algorithm 1, blocks are denoted by `Bi`.

5.3 Preparing a driver

A driver produces a sequence of test vectors, previously selected or generated data. Driver, testport and code item are combined into a single executable whose execution proceeds to call the code item through the testport using the driver's test vectors. Using a simulated annealing algorithm as the driver (see Section 4), the configuration stage includes selecting the right fitness function and setting up the search parameters, e.g. initial temperature.

5.4 Executing the testport

Each execution of the testport produces a corresponding trace including at the traversed instrumentation points, and thus the covered code blocks. Additional implementation-specific information may be collected regarding the behaviour of the driver or the underlying platform, such as blocks' execution times. Each such trace can be linked back to the specific test vector that drove the execution.

Example: *The execution of the test vectors defined in Section 4 for Algorithm 1,* `{A: 1.0, B: True, C: 4}` *and* `{A: 1.0, B:False, C: 0}`*, produces two execution traces, respectively [B1, B2, B3, B5, B6, B6, B6, B6, B8, B7] and [B1, B2, B4, B7].*

5.5 Processing execution traces

The execution traces can then be processed and compared against the structure of the code to identify covered and missing code blocks both in the code item and in its sub-programs. The TACO process can then be repeated; by generating a testport and driver for the uncovered items, the tool can iteratively build complete block and timing coverage of the system.

6 PORTING THE TACO FRAMEWORK

The limited availability of the target platform for a system, due to costs or a limited number of units for testing, may hinder the capability to perform tests. Target-based testing may also be expensive and time-consuming due to resource constraints on safety-critical embedded platforms. There is therefore a strong requirement to reduce the time spent running on the target platform which contradicts the use of a runtime-based search algorithm. Limits on memory and available communication channels may further restrict the class of applicable algorithms for automated search and the capacity to extract valuable information during the process.

The BCHLr heuristic (Section 4) has no dependency on the underlying platform. It relies on the executed path through a test item under analysis, as defined by source-level blocks of code. Similarly, testports and the input vectors are generated to be platform-independent. The range, type and number of inputs in the analysed software are known before compilation (Section 3). The set of values each input can take, and their impact on the program flow, are independent of the underlying platform[1].

[1] Different platforms may use different precisions for floating point values, but their range is still bounded and representation precision should have no bearing on the executed path in the considered software.

The instrumentation routines capture the execution of instrumentation points in the code and feedback to the BCHLr heuristic on the executed paths through the code. The timing data associated to instrumentation points only needs to be captured on the Rolls-Royce platform (Section 3.1), and is easily omitted from development platforms. Instrumentation primitives are simply replaced by writes to an allocated buffer in memory; the instrumented code itself is left unchanged. Like on the VISIUMCORE, the coherence between the source and compiled code is guaranteed by the instrumentation process, and thus coherence between platforms.

The Input/Output primitives provide the interface to extract information from the ongoing execution of a code item, such as the exercised test vectors or more importantly the execution trace. General purpose platforms offer a wide array of channels for communication. Ports of the framework for the ARM Raspberry Pi3, using a Real-Time Kernel and general purpose desktop platforms output a wide variety of statistics on the ongoing search through simple output files.

6.1 Reducing on-target testing

As a proof of concept, we now consider an application of the information extracted from a host platform to improve the coverage achieved on target while reducing the number of tests on target. This is a further illustration of the compositional nature of the TACO framework to define new analysis processes. The use of a general purpose computing platform, as opposed to the VISIUMCORE embedded processor, offers the opportunity to extract more information from the running search algorithm. In particular, during the execution of a testport we extract the inputs exercised on the code item under test at each stage of the search algorithm. Exercised inputs can thus be matched against the instrumentation points they exercise in the tested item.

Using a simple heuristic, we reduce the set of exercised inputs and test items on the general purpose platform, to one that can achieve the same level of coverage at a lower cost. The selected inputs are replayed by using a driver that iterates through the test vector selection for the relevant code items. The selection first reduces the set of exercised items, by selecting the least observed instrumentation point and picking the item that reaches it most often. The relevant inputs within each selected item are then selected to reach the instrumentation point. The process is outlined in Algorithm 3. The selection is performed first at the item level, instead of the run level, to reduce the complexity of the problem; millions of iterations, and the instrumentation points they reach, would need to be loaded and compared to reach an optimal solution that minimises the overall number of required runs.

7 EVALUATION

We applied the TACO framework on a range of items from a Rolls-Royce (RR) aircraft engine controls system. The evaluation includes two different targets to compare their behaviour and support the claim that the heuristics are platform-independent. This includes a standard desktop computer, *i686*, and the VISIUMCORE, *VISIUMCORE* (Section 3.1). TACO was applied in bulk, with the full process being repeated for each test code item. We further include the Ran fitness function which always accepts new solutions.

Algorithm 3 Selection of test code items and inputs to achieve target coverage from observations.

```
1  Replay_Data := {}
2  Replay_Points := {}
3  Sort Observed_Points from least to most observed
4  for P in Observed_Points
5    if P not in Replay_Points
6      Sort Test_Items from most to least covering P
7      for I in Test_Items
8        for T in Traces(I)
9          if P in Points(T)
10           Add (I, Inputs(T)) to Replay_Data
11           Add Points(T) to Replay_Points
12           Next P
```

7.1 FADEC system

The industrial test code used for the analysis was taken from an unmodified Rolls-Royce engine control system and has been designed and verified according to DO-178C standards as a level A package [17]. The models required for the generation of the TACO testports were made available for a selection of 1800 test code items. They cover a broad section of the engine control system from the tasks dispatched by the scheduler (250+), down to small utilities function. Some items contain complex constructs, input dependant loops or infeasible paths, whereas other items are more simplistic and contain fewer branches and simpler constructs. High level functions are also included, which further call other items. This is important to evaluate the scalability and performance of any automatic analysis.

Some items further include a number of internal (or static) 'state variables' carried forward to the execution of the test code item from its previous execution. The Rolls-Royce code items are taken from a control system which incorporates a large amount of feedback, this means different test iterations are influenced by previous system state, set up by previous test iterations. An example of this is a fuel management system could determine how much fuel is left not only based on a level sensor but also based on an earlier trusted level and by continuously using the fuel flow rate to get a *predicted value* of how much fuel has been used since the earlier trusted level. This type of calculation can be used for both situations where a *current value* of a parameter is hard to obtain or for validating using dissimilar sensed values the current value. It may be that some Fault Accommodation Code can only be reached if the current and predicted values differ by x%.

Hence, state is an important feature that cannot be ignored. Each test may thus execute the state from previous test iterations and has a significant effect on the current path. This emphasises how difficult it can be to manually generate test cases that provide sufficient coverage for MBTA. Each state variable contained within each test code item is modelled as a potential input by the testport. However in this experiment they are only randomly exercised to let the system progress naturally during some successive runs. This proved to be the most efficient configuration across all heuristics. Not exercising the state as inputs restricts the set of paths that are reachable by the search algorithm. Conversely, always enforcing the inputs on the internal state ignores the natural rate of change for some values and can prevent the execution of specific paths in the code.

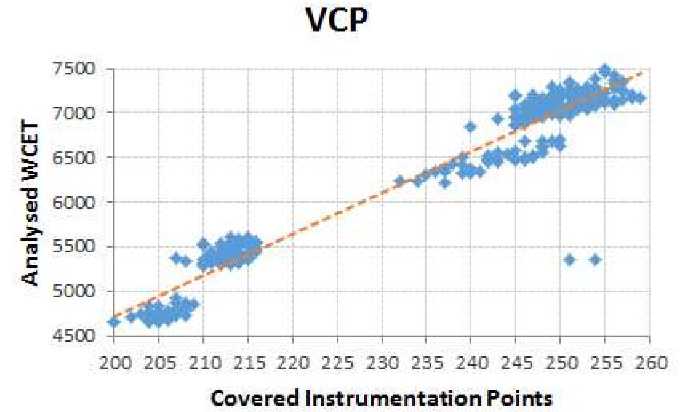

Figure 2: Relationship between covered instrumentation points and analysed WCET on the RR platform.

7.2 Correlation between timing and coverage

The timing of a block on the VISIUMCORE platform is impacted neither by the prior execution, nor by the inputs applied to the software. This creates a correlation between code coverage (observed blocks) and WCET estimates computed through MBTA (combined timings of blocks). We applied TACO to a small subset of code items in the case study, using both BCHLr and Ran heuristics, and fed the results through the RapiTime tool to produce a WCET estimate.

Figure 2 illustrates the correlation for one such code item, VCP. When a new path and thus new instrumentation points are covered, the analysed WCET increases in consequence. The gap in the number of instrumented points stem from a hard to reach path in the code item, once reached many other branches are discovered and explored by the algorithm. Because of the instrumentation profile use during the case study, i.e. one instrumentation point at the beginning and end of each function, even straight line code may hold more than one instrumentation point.

We focus on code coverage as a metric in the rest of the evaluation. Coverage still allows for a broad view comparison between the performance of different search heuristics across wildly different code items. Due to the scope of the experiments, collecting timing data on target for all code items and considered configurations could take a considerable amount of time[2].

7.3 Framework scalability

The TACO framework aims at providing an automated cross-platform framework for driving the execution of test code items, that is the ability to take a given system build and automatically exercise its individual components. We first evaluate the scalability of the framework on the test items available as part of the case study. Figure 3 captures the portion of the evaluated items for which a testport was generated and successfully driven for each configuration.

Overall, more than 95% of the 1800 test code items have been successfully analysed through the framework. Some items are not amenable to testport generation in the absence of some contextual

[2] 22 million test vectors were tested by the BCHLr heuristic on the i686 platform in less than 48 Hours, where 2000 iterations take an average of 1 Hour on target [14] because of data upload/download overheads.

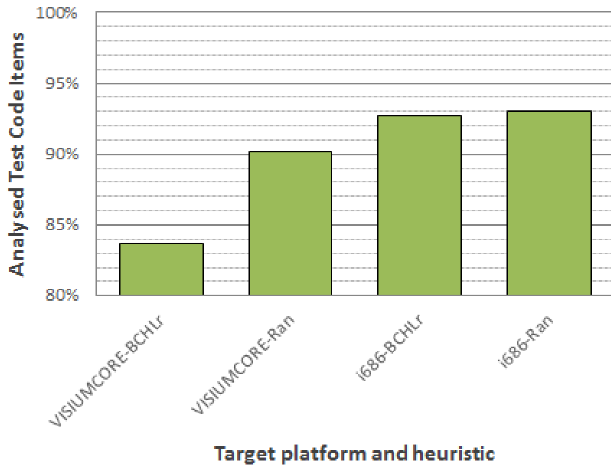

Figure 3: Test code items successfully analysed through TACO

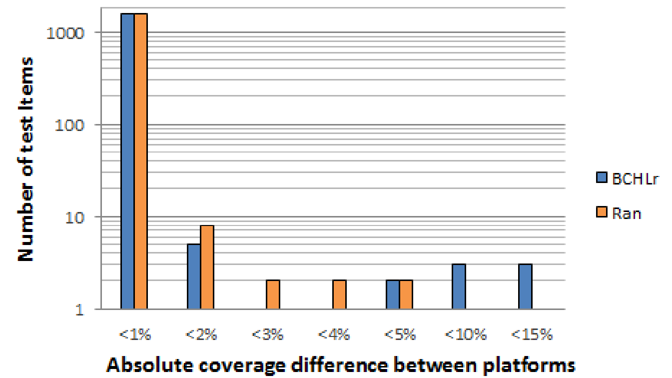

Figure 4: Absolute difference between the coverage achieved on different platforms for the same heuristic and test code item.

knowledge, e.g. look-up tables containing system-specific constants, which impact the size of input arrays. Information beyond the test code item model is then required to generate a testport.

On the VISIUMCORE, less items could be processed using the Ran or BCHLr heuristics, respectively 90% and 83%. While the heuristics applied to drive the testports are item and platform-independent, their overheads and memory complexity relate to the item under analysis. As an example, the memory required to store the history for the BCHLr heuristic increases with the number of inputs and instrumentation points in a test code item. For resource constrained targets, such as the RR embedded platform, those structures may not fit in memory of embedded platform and only simpler heuristics, like Ran, can reliably run on target.

7.4 Portability of the heuristics

The heuristics evaluated in this work do not rely on the behaviour of the underlying platform to drive the search algorithm. Instead, they rely on the exercised paths through the program as captured at the source level. We ran each configuration of test code item and heuristic on both the desktop platform and the RR platform. Figure 4 presents (using a log scale on the y-axis) the resulting difference for each configuration in achieved coverage, i.e. the impact of applying the configuration on a different platform. The items that cannot be analysed on both platforms have been omitted to ensure a fair comparison. This includes as an example issues related to BCHLr history memory occupancy on the RR platform. Figure 5 further presents for each item under the BCHLr heuristic the coverage achieved on both platforms. There are only marginal differences between runs of the heuristics on different platforms with more than 99% of configurations exhibiting the same behaviour. The remaining differences amount to differences in the pseudo random number generator between platforms and thus differences in the generated test vectors.

7.5 Scalability of the heuristics

We now consider the performance of the test generation framework in terms of code coverage. Figure 6 presents the overall coverage

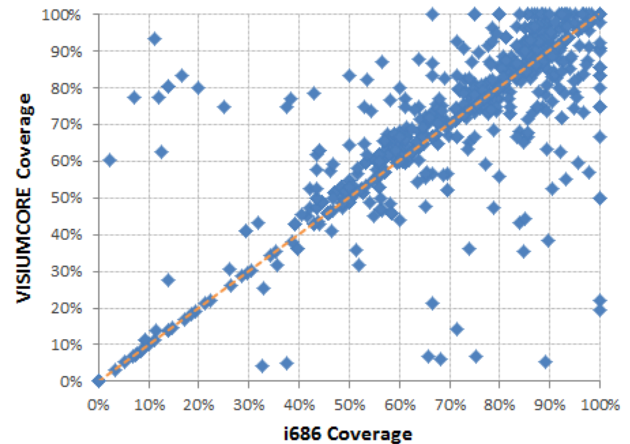

Figure 5: Comparison of the coverage achieved by the BCHLr heuristic on VISIUMCORE and i686 platforms.

achieved by each heuristic across the whole set of test items on the different platforms. The results consider for each item solely the blocks covered during its run of the analysis. It captures the number of items which coverage after execution of the framework falls within a given coverage range. The further to the right a range ends, as compared to a different configuration, the more items achieve the same or higher coverage. As an example, more than 1100 items have a coverage > 70% from BCHLr on the i686 platform. Figure 6 also captures the items that fail to run, flagged with a 0% coverage, as an example using BCHLr on the VISIUMCORE.

Overall, the BCHLr heuristic on the i686 platform allows for marginally greater items' coverage at almost all given levels. However, those are obtained at a much greater memory and time complexity than its Ran counterpart. Very restricted platforms cannot accommodate for the size of the Branch History of some code items. On the VISUMCORE, the lower number of items compliant with BCHLr requirements reduces its performance. Ran also tends to converge after much less moves as it does not benefit from reheating in the absence of rejected solutions. This raises the issues of diminishing returns as more iterations are required to reach the

less likely paths, w.r.t. the input vector values distribution, in the application.

The results in Figure 6 only provide a partial picture of the coverage of the system achieved by the TACO framework. Consider items A and B such that A calls B during its execution. Some blocks of B may be hard to reach when called from A, resulting only in partial coverage of A during tests. If those blocks are covered when B is tested in isolation then their results can be merged in the analysis of A.

We thus extend the computation of the coverage achieved for a test code item to include the whole set of blocks covered during the analysis of the system, not only those encountered during its analysis. The resulting merged coverage is presented in Figure 7. For the sake of brevity, we omit the inclusion of results combined across other platforms and/or heuristics.

Looking at system-wide results, more than 1100 items have been fully covered by the automatic analysis (1132 for Ran, 1326 for BCHLr and 1335 by combining both heuristics). Although some items cannot be executed under specific configurations, credit for the functions they call can still be taken and used to estimate their behaviour. On the i686, while the BCHLr heuristic covers more items (1326), additional items are covered by Ran resulting in a combined coverage of 1335 items. Combining results across platforms and heuristics, 1375 items are fully covered showing that the heuristics are complementary and more effort results in higher coverage albeit with diminishing returns. This further illustrates a benefit of a tool that builds the coverage of a system, from the top-level tasks down to the small utility functions.

7.6 Reducing on-target testing

The availability of multiple platforms to evaluate the coverage achieved by different heuristics and configurations using the TACO framework would be of little value if they were not applicable to the target RR platform. We now explore how to utilise information extracted from the host platform to reduce on-target testing.

As a proof of concept, we applied the heuristic defined in Section 6.1 to the observations made on the i686 platform through the BCHLr heuristic which encompassed 22,000,000 executions across 1775 test code items. The coverage achieved by this configuration is presented in Figure 7.

To achieve the same level of coverage on other platforms, only 6131 tests across 658 items need to be replayed. This is an average of 9 executions required per item with a maximum (respectively minimum) of 168 (resp. 1) executions. The overall replay experiment took less than a couple of days to complete using a target simulator to extract coverage. In comparison, 2000 iterations of a single code item took on target an average of 1 Hour in previous setups [14]. 658 code items could thus be tested in the timespan 3 (52 including collection on the i686 platform) code items can be run natively through TACO on target, while providing higher coverage results. This assumes TACO could be executed with the code items within the memory limits of the target.

Figure 8 presents the results of applying the selected tests on both the desktop host and RR target platforms. Minor differences in the achieved coverage of the original and replayed runs illustrate the loss of precision in the inputs values extracted during the original runs. Difference between platforms also amount to the lower number of items supported on the VISIUMCORE with some items selected by the heuristic not executed, e.g. when the set of test vectors does not fit in memory. For further details, see Figure 3.

8 CONCLUSION AND FUTURE WORK

This paper has described an automatic test generation framework which can help produce good coverage and thus WCET estimates, when supported by a MBTA, early in the software lifecycle. A key aspect of the approach is a search algorithm that is platform independent as:

(1) There is no dependency on timing information.
(2) The tests produced by the search algorithm on a host, or low-cost platform, can then be applied on the target.
(3) The host-based testing provides meaningful results.

The long-term aim of our research and development is to use low-cost platforms, including desktops, to:

(1) Provide estimates early in the life-cycle
    (a) understand the timing characteristics of software, e.g identify the longest parts of the software.
    (b) predict the values of HWM and WCET using information about how host-based coverage relates to timings on target.
(2) Efficiently generate the set of tests needed for when the target is available. Ultimately the target-based testing will consist of:
    (a) The best N different test cases found on the low-cost platform will be replayed on the target.
    (b) Where possible BCHLr will be applied to the target to validate the results and check that no new information is found.
    (c) If BCHLr is not possible, then the search algorithm is performed with the next best fitness function.
(3) Changes to the system could be quickly assessed in terms of understanding the regression testing needed on the target.
(4) On existing systems, to quantify the confidence that is obtained from the proposed system compared to our existing manually generated tests.
(5) On future systems, to quantify the confidence of the testing performed so that the developer knows when to stop testing and as part of the higher-level design and analysis of the mixed-criticality scheduling.
(6) Evaluate alternative drivers and configurations before deployment on target.

On the target, the generated tests could then be used to find the HWM and analysed WCET and some limited search-based test generation could be performed. (Limited by the size of the buffers on the target and the ability to offload via the communications the information output.) This testing would give the final values for HWM and analysed WCET and validate the earlier assumptions, i.e. the correlations between coverage and timings on the low-cost platform and the target.

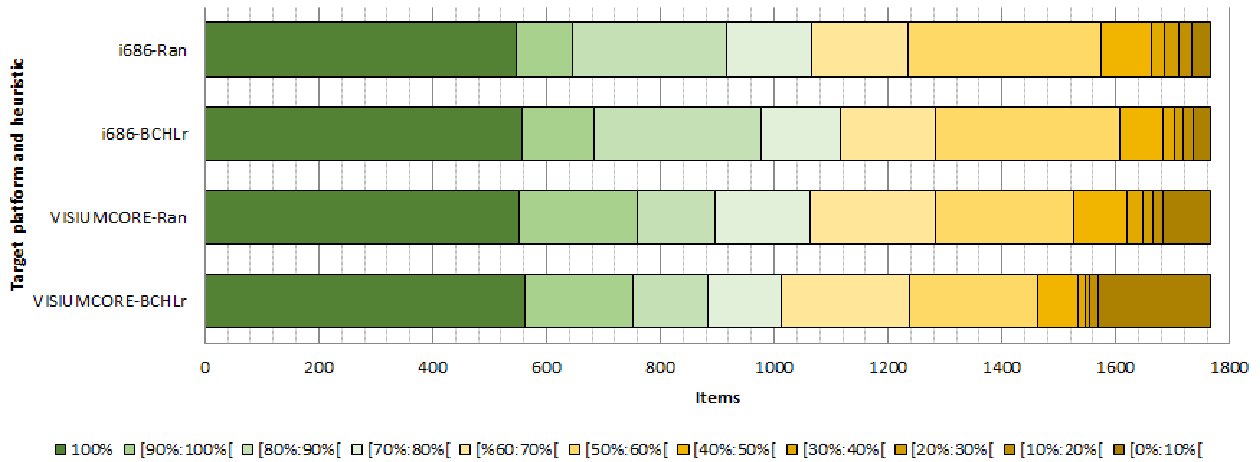

Figure 6: Block coverage achieved on the test code items by the heuristic

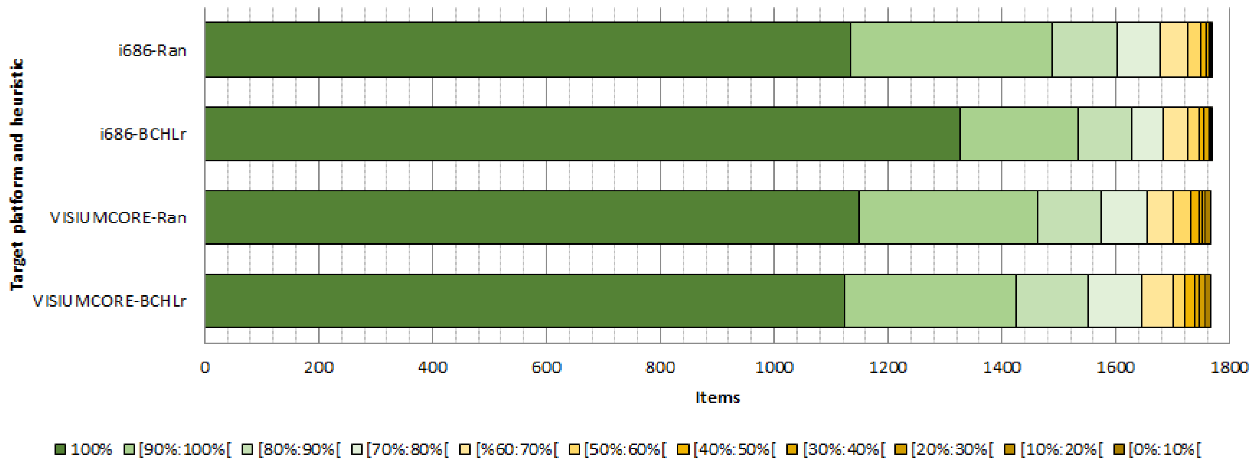

Figure 7: Block coverage achieved on the test code items after merging results across test items

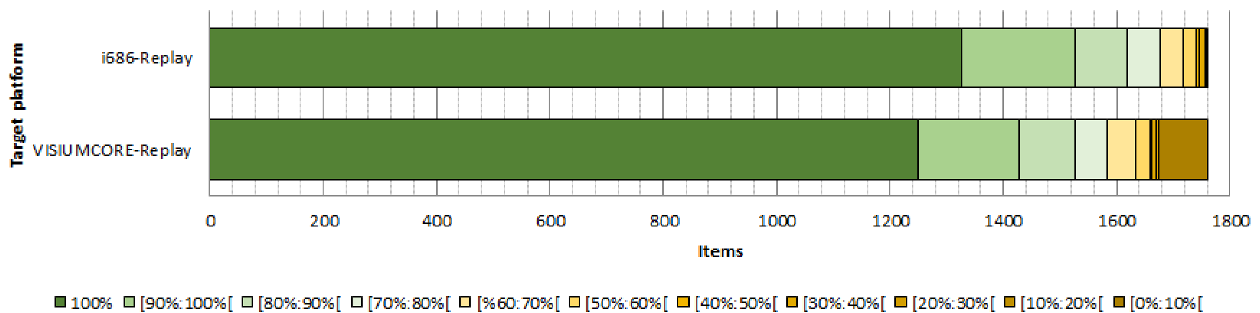

Figure 8: Block coverage achieved on the test code items after replaying selected tests and merging results

ACKNOWLEDGMENT

This work was funded by the INNOVATE UK funded Automated Timing framework of Interference in Critical Systems (ATICS) project (KTP010577).

The authors wish to thanks Stuart Hutchesson and Mike Bennett for their valuable opinions and review comments throughout the course of this work.

REFERENCES

[1] Iain Bate and Usman Khan. 2011. WCET analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering* 16, 1 (2011), 5–28.

[2] Sven Bünte, Michael Zolda, and Raimund Kirner. 2011. LetÃćÂĂÂŹs get less optimistic in measurement based timing analysis. In *Proceedings of the 6th International Symposium on Industrial Embedded Systems*.

[3] Sven Bünte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. 2011. Improving the confidence in measurement-based timing analysis. In *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 144–151.

[4] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quinones, and Francisco J Cazorla. 2012. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*. 91–101.

[5] Jean-François Deverge and Isabelle Puaut. 2005. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on WCET Analysis*, Vol. 5.

[6] Stewart Edgar and Alan Burns. 2001. Statistical analysis of WCET for scheduling. In *Proceedings of the 22nd Real-Time Systems Symposium*. 215–224.

[7] S. JimÃĺnez Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. 2017. Open Challenges for Probabilistic Measurement-Based Worst-Case Execution Time. *IEEE Embedded Systems Letters* 9, 3 (Sept 2017), 69–72. DOI: http://dx.doi.org/10.1109/LES.2017.2712858

[8] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*.

[9] Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. 2009. Statistical-based WCET estimation and validation. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*.

[10] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. 2008. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *Proceedings of the 20th International Conference on Computer Aided Verification*. 209–213.

[11] B Jones, H Sthamer, X Yang, and D Eyres. 1995. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management*. 435–444.

[12] Usman Khan and Iain Bate. 2009. WCET analysis of modern processors using multi-criteria optimisation. In *Proceedings of the 1st International Symposium on Search Based Software Engineering*. 103–112.

[13] S. Kirkpatrick, D. Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.

[14] Stephen Law and Iain Bate. 2016. Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis. In *Euromicro Conference on Real-Time Systems*.

[15] Stephen Law, Mike Bennett, Ivan Ellis, Stuart Hutchesson, Guillem Bernat, Antoine Colin, and Andrew Coombes. 2015. Effective Worst-Case Execution Time Analysis of DO178C Level A Software. *Ada User Journal* 36, 3 (2015), 182–186.

[16] Rapita Systems Ltd. 2018. RapiTime Explained: White Paper. (2018). http://www.rapitasystems.com/downloads/brochures-white-papers/rapitime-explained

[17] RTCA. 2011. DO-178C - Software Considerations in Airborne Systems and Equipment Certification. (2011).

[18] Stefan Stattelmann and Florian Martin. 2010. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*.

[19] Nigel Tracey, John Clark, Keith Mander, and John McDermid. 1998. An automated framework for structural test-data generation. In *Proceedings of the 13th International Conference on Automated Software Engineering*. 285–288.

[20] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. 1997. Testing real-time systems using genetic algorithms. *Software Quality Journal* 6, 2 (1997), 127–135.

[21] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. 2009. Measurement-based timing analysis. In *Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 430–444.

[22] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem; overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (2008), 36.

[23] Nicky Williams. 2005. WCET measurement using modified path testing. In *Proceedings of the 5th International Workshop On Worst-Case Execution-Time (WCET) Analysis in conjunction with the 17th Euromicro International Conference on Real-Time Systems*.

[24] Nicky Williams and Muriel Roger. 2009. Test generation strategies to measure worst-case execution time. In *ICSE Workshop on Automation of Software Test*. 88–96.