Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis

Stephen Law*, Iain Bate†
*Rolls-Royce Control Systems, Birmingham, UK
Email: stephen.law@rolls-royce.com
†Department of Computer Science, University of York, York, UK
Email: iain.bate@york.ac.uk

***Abstract*—Establishing Worst Case Execution Times (WCET) using Measurement-Based Timing Analysis (MBTA) is only effective if we have reasonable confidence that we have fed the worst case execution trace into the analysis. Therefore for certification, the quality of these traces is of paramount importance. This paper aims to investigate how search algorithms can be used to automatically, and reliably, generate test cases so that appropriate execution traces are available to support MBTA. The work carried out in this paper uses a standard search algorithm and created a number of fitness functions to target the generation of 'good data'. The results are then input into a commercial measurement-based WCET analysis tool. The new fitness functions focus on achieving a combination of full branch coverage and a high number of loop counts, or partial path coverage, however are shown to achieve reliable approximations of the WCET particularly when combined with an MBTA tool. The code items used for the analysis included off the shelf benchmarks, as well as industrial safety-critical aircraft engine control software.**

I. Introduction

Analysis of a system's WCET is a key technique in all real time systems, particularly in a safety-critical environment. There are two key methods of WCET analysis, static analysis and measurement-based.

Static analysis takes the code of the System Under Test (SUT), analyses the possible paths through the code, and by modelling the target hardware; calculates which path through the SUT will produce the WCET, as well as the actual WCET. Alternatively some approaches define an upper bound on the execution time of the SUT [1]. The analysis gains from being able to fully examine the full set of paths through the SUT. However the primary drawback of static analysis is the technique's reliance on accurate processor models. As developers look to use ever more complex processors the complexity of these models increases accordingly [2].

MBTA approaches rely on measuring the execution of the SUT to provide measured times which are then used to derive WCET bounds. The advantage of this approach is that times can be derived from the actual target hardware, with no reliance on complex timing models. However the technique suffers from the fact that the software must be executed on the target hardware (or equivalent cycle accurate simulator) with a sufficient level of coverage to provide accurate results.

Traditionally one measurement technique used in industry has simply timed the SUT as it is executed as part of standard software verification tests. The maximum observed execution time (MOET, or High Water Mark - HWM) is then taken forward with the addition of a safety bound (defined through engineering judgement) to produce an acceptably sound WCET [3]. One of the biggest risks with this approach being that the testing may not drive the worst case path.

Rolls-Royce Controls Systems develops safety-critical aircraft engine control systems. RapiTime, an MBTA tool [4], is used for analysing the WCET of software targeting our in-house processor. As the processor executes measurements are taken throughout the code, using specially inserted instrumentation points, the measurements taken and fed into the tool are obtained during the execution of low-level test scripts [5]. The issue with this approach being that results cannot be obtained until these scripts have been produced.

Certification is driven by having high confidence that the requirements of a system are met. For these reasons, it is not required that know the Actual WCET (AWCET), but that we have confidence we are close to it so that the likelihood of a deadline overrun is minimised.

The confidence in the HWM, or RapiTime WCET (RWCET) is reinforced by an argument about coverage. Coverage is a key part of justifying sufficient testing in most certification standards especially the one that our industry uses, i.e. DO-178C [6]. In terms of the WCET, Betts et al [7] identified coverage metrics for MBTA, however these metrics ultimately equated to achieving more than state coverage, and so as software complexity increases they become virtually impossible to achieve. At present the available literature does not indicate these metrics have been applied to an industrial scale project. Instead, in this paper it is argued that branch coverage is the absolute minimum and path, and hardware state, coverage is ideal. However as achieving path or state coverage is generally infeasible in an industrial scale project, in reality it is argued that maximising loop bounds and achieving path coverage for individual functions is preferable. The contributions of this work are to:

1) Extend the state of the art approaches for Automatic Test Case Generator (ATCG) to more reliably achieve a WCET close to the AWCET within a finite amount of time - reliably is defined in the paper as not a rare event but instead a result close to the actual maximum most times. As the AWCET is not known in practice, the

AWCET is taken to be the WCET over the vast amount of trials (i.e. all the repeated trials of each of the different ATCG variants) that are performed.
2) As part of 1) an achievable coverage metric is derived for measurement-based WCET analysis.
3) Present a comprehensive evaluation not only based on our real engine control systems but also using examples from the Mälardalen benchmark set [8].

This paper ultimately presents a new fitness function to support a standard search algorithm, the algorithm focuses on achieving full branch coverage, and maximising loop counts. It is the product of an assessment into what input data is actually required to drive an MBTA tool. An important issue explicitly considered is that many real systems carry state from one execution of a task to another, for example feedback-based systems, and hence the amount of state that has to be handled is much more than typical benchmarks previously considered.

The structure of this paper is as follows; Section II introduces related work into the field, focusing on MBTA and ATCG. Section III then introduces a set of search algorithm parameters designed to automatically produce input data for an MBTA tool. Finally Sections IV and V describe an experiment designed to measure the effectiveness of each search algorithm.

II. Related Work

This section presents a study of the existing work in the field of MBTA and ATCG.

A. Measurement-Based Time Analysis

MBTA techniques aim to simplify the problem by either reducing the input space, or by extrapolating the results obtained from a sample of results. Deverge & Puaut [9] proposed a solution avoiding the need to exercise the SUT fully which is to measure all paths of a program on a segment by segment basis. In a similar vein Stattelmann & Martin [10] presented an MBTA tool that also breaks the SUT into a number of easily traceable segments; the WCET is then constructed as a mathematical equation across each segment. Ultimately the risk with both solutions is their scalability; as the complexity of the system increases the number of sections the code is broken down into would increase accordingly. For a large industrial scale project this could lead to tens of thousands of functions all being analysed to provide path coverage, or all producing a context-sensitive WCET equation, and so the processing, or engineering, effort required would be 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 SUT down through the use of source code instrumentations, in contrast to [10] however the analysis is performed at the block level, and depending on the target hardware, does not require path coverage. Each instrumentation triggers the output of a tracepoint. The software structure and maximum measured time for each block is then used to construct a WCET. Crucially in an industrial context the tool is able to analyse the SUT as verification test scripts are executed. Depending on the target hardware, and how loops are handled, the tool only requires that the test scripts used provide full branch coverage through the code.

Measurement-Based Probabilistic Timing Analysis (MBPTA) was first proposed by Stewart & Burns [11]. This was later extended by Hansen et al [12] and Cucu-Grosjean et al [13]. The basis of these techniques is the use of Extreme Value Theory to fit an appropriate distribution to the observations captured. The WCET is then extracted from the distribution for a chosen level of probability that it is exceeded. The problem is that in order to provide reliable results the input data fed into the tool must be independent and identically distributed, which in practice is hard to achieve. The code coverage required, in some cases state coverage, makes the problem even harder to achieve.

Ultimately all measurement techniques have the same basic requirement, that they require good input data to produce good results. The previous work in the field has focused on generating results with a given known good data set, or in the case of [13] using hardware randomisation to force hardware to make a data set *good*.

B. Automatic Test Code Generation

Wegener [14] and Tracey [3] both illustrated how search algorithms could be used for test data generation, particularly with regard to applications that require coverage beyond statement coverage.

Wegener's early work [14] built off Jones et al [15] 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 SUT. One of the targeted analyses being the analysis of the WCET. The work has been targeted toward safety-critical systems using strongly typed Ada [3]. 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 [16] introduces an MBTA tool designed to calculate safe WCET bounds of safety-critical software. The tool uses a combination of static analysis, and dynamic measurement of the SUT in order to compute safe WCET bounds. The tool statically analyses the feasible paths through the code, then uses search algorithms to identify test vectors to execute each path. This is achieved through a combination of test data reuse, random search, genetic algorithms and finally model checking [16]. 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. So unfortunately the compromises required to use the tool are significant, and would not be acceptable in an industrial environment.

Williams [17] 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 [18] 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 [19] examined the effectiveness of using model checking [20] 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 [21] which combines the results produced with a genetic algorithm, which then aims to identify larger execution times. One drawback is that the tool analyses software that has been simplified to ensure each decision point relies on only a single variable. This may not be appropriate to an industrial program where large amounts of generic code are carried forward to future programs. Also the tool's use of model checking risks the tool's portability to larger, more complex functionality. These aside the tool shows some of the most advanced work in the field of MBTA data generation.

Khan and Bate [22] introduce the idea of incorporating multi-criteria optimisations into a search based WCET analysis tool. The method adopted used a number of fitness function parameters in order to attempt to drive the worst case path, these included advanced processor features known to cause larger WCET values, such as cache misses, but 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. However the paper focused on a number of processor, or software, features that are not necessarily present in safety-critical systems and also didn't consider coverage which is of importance to certification.

This paper is concerned with using search algorithms to generate good data for input into MBTA tools. This allows the search algorithm to be focused on a smaller, more manageable search space that delivers the good input data' required by the MBTA method adopted. The work differs from previous approaches, such as [16] and [21] as firstly the fitness functions used have been specifically tailored to target the type of data needed by the MBTA tool. Secondly the analysis places no restrictions on the software under test, and has been investigated on a processor, and software set, taken directly from industry. This includes software that features a large amount of previous software state, which significantly increases the search space; to our knowledge this has not been investigated by the available literature.

A comparison is made to the approaches suggested by Tracey [3], Jones [15] and Wegener [14] as well as the better performing fitness functions suggested by Khan [22].

III. Investigating a Combined Approach to WCET Analysis

The investigation aimed to identify how effectively a basic search algorithm could be at generating data for a hybrid MBTA tool, in this case RapiTime [4]. The work builds off the current industrial setup, as described by [5]. The study used a number of fitness functions in order to identify how different targets alter the results obtained.

A. Algorithm Objectives

As the search algorithm executes on the target timing measurements are taken across the SUT. 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. It is to obtain high code coverage across the SUT to ensure the RWCET approaches the AWCET.

The following objectives are derived based on the overall objective of the end user. That is reliable, automatic and consistent estimation of the WCET in a reasonable cost effective time-frame.

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. This objective is assessed by recording the highest execution time observed during test execution, prior to input into RapiTime.
2) Consistently the highest iPoint coverage - If the test has not achieved good iPoint 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 RWCET. The objective is assessed overall by comparing the number of tests that achieve greater than 90% of the total iPoint coverage.
3) Consistently large RWCET results. This is the ultimate aim of the algorithm, to produce the largest possible RWCET result. This objective is assessed by comparing the distribution of results produced by each fitness function, with particular attention paid to comparison against the ET fitness function. A statistical analysis is then used to identify whether the results provide a large enough sample to indicate significance.

B. Search Algorithm Setup

The search algorithm used for the analysis is a derivative of the simulated annealing algorithm, originally presented in [23]. The basic algorithm is shown in Algorithm 1.

The simulated annealing algorithm was chosen over other algorithms, such as a genetic algorithm, because of its ability to narrow down on a good solution, while also searching over a large part of the search space. Although the key to this work is the fitness functions proposed, there is no reason why these fitness functions couldn't be used to drive a genetic algorithm.

Algorithm 1 Simulated Annealing

1: $Temp$ = [0.01, 0.1]
2: **while** NOT StoppingCriteria() **do**
3: $NewSolution$ = GenNewSolution($CurrSolution$)
4: $Fitness$ = FitFunc(TestCode($NewSolution$))
5: **if** random(0..1) $<$exp($Fitness$ / $Temp$) **then**
6: $CurrSolution = NewSolution$
7: **else**
8: ignore new solution
9: **end if**
10: $Temp$ = CalculateNewTemp($Temp$)
11: **end while**

On each iteration the $GenNewSolution$ function pseudo-randomly selects a new input solution to the function under test, this solution is generated from the previous solution, with only a minor change to a single randomly selected variable. $FitFunc$ is then used to assess the new solution's fitness, which is accepted, by the if statement on line 5, if an improvement, or pseudo-randomly selected if a degradation. As the test progresses the pseudo-random selection of worse solutions will reduce, as controlled by $Temp$. Finally $StoppingCriteria$ will end the search once no solutions have been accepted in the previous third of the test run (with a basic minimum of 1000 iterations).

One modification from the original algorithm suggested by [23] has been made, that is if no solutions are accepted after 200 iterations, then the temperature is increased to reheat the search [24]. This reheating schedule was shown to avoid the simulated annealing algorithm being caught in a local minima, which is regarded as one of the risks with the algorithm. In order to allow comparison this standard algorithm was used for all tests code items and fitness functions.

C. Fitness Functions

The analysis uses a standard search algorithm, but seven different fitness functions, partitioned into five groups, have been defined and compared. Based on the MBTA requirements detailed in Section II-A fitness functions have been defined to attempt to produce optimum data for MBTA tools. Two further fitness functions have then been defined based on the previous ATCG techniques for comparison.

The fitness functions defined were broken down into the following groups:

- *Random* - All solutions are accepted automatically.
- *Maximum Execution Time* - The function aims to increase the execution time of the SUT. Proposed by Tracey [3], Jones [15] and Wegener [14].
- *Branch Coverage* - The fitness function aims to execute all branches through the code.
- *Maximum Loop Counts* - The fitness function aims to maximise the number of iterations of each loop through the code. Proposed by Khan [22].
- *Changes in Execution Time* - The function aims to identify potentially different paths not seen before, each path is identified by its execution time.

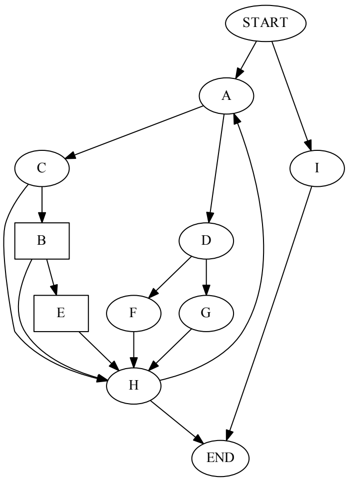

Fig. 1. Example CFG. The rectangular boxes refer to blocks that have not been executed, the oval boxes refer to blocks executed during this test.

1) Current Approach to WCET: **Execution Time (ET)** is designed to attempt to identify the largest execution time possible. As each new solution is executed its operation is timed, the current execution time is then assessed against the previously accepted execution time. This is shown in Equation (1), where $CurrTime$ is the time of the current solution, $PrevTime$ is the previously accepted best solution and $Fitness_{ET}$ is the fitness calculated. The subtraction of one from the time difference ensures that an identical execution time is not viewed as an improvement.

$$Fitness_{ET} = \frac{CurrTime - PrevTime - 1}{PrevTime} \tag{1}$$

2) Branch Coverage: **Branch Coverage (BC)** assesses the fitness at every branch through the current path, the branch's fitness is calculated as the normalised sum of the number of edges out of the branch. The solution fitness is then calculated as the average fitness of all branches on the current path. For example referring to Figure 1 if the current solution's path includes block C, (or the previously unseen blocks) B or E then the fitness calculated will be significantly higher than if the path traverses through blocks D, F, G, H or I. So the algorithm is weighted more towards the full execution of each branch through the code, and is weighted less towards path coverage.

Bünte et al [19], proposed the use of Modified Condition/Decision Coverage (MCDC) to provide WCET coverage. However we argue that MCDC is not necessary in this context as MCDC would not offer further refinement of the results over branch coverage. Ultimately this would lead to a harder search, without providing better results. For example referring to Figure 1, we do not care how we made our decision at block D, only that we executed both blocks E and F. If the decision at D is based on a large number variables (N), then the search space would increase from 2, to 2^N.

$$CurrFitness_{BC} = \frac{1}{B_p}\sum_{b=0}^{B_p}\left(\frac{1}{E_b}\sum_{e=0}^{E_b} unseen(e)\right) \quad (2)$$

Equation (2) shows how the fitness for the current solution is calculated, where $unseen$ is an array which records each edge which has not been executed, E denotes edges from this node and B_p denotes branches on the current path. The division by B_p ensures the result is normalised for being input into Line 5 of Algorithm 1.

Branch Coverage History (BCH) uses the same basic fitness calculation as BC defined by equation (2). However as each branch through the current solution's path is analysed, the input vector used to drive the current solution is stored against that branch. If after fifty iterations the solution has been rejected continuously then the set of outgoing edges that have not been fully executed is examined, and one 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.

$$Solution\_Array[b] = CurrSolution, b = 0..B_p \quad (3)$$

$$NewSolution = \begin{cases} \text{GenNewSolution}(CurrSolution) \\ \qquad \text{if } Reject <= 50 \\ Solution\_Array[\text{rand}(B_{NFE})] \\ \qquad \text{if } Reject > 50 \end{cases} \quad (4)$$

Equations (3) and (4) describe how the algorithm operates; on each iteration the current solution ($CurrSolution$) is recorded against each branch found upon the current path, as denoted by B_p. Equation (4) replaces line 3 of Algorithm 1; on each iteration if the previous fifty solutions have been rejected then next solution ($NewSolution$) is set to equal a solution taken from the $Solution\_Array$. The array value chosen is selected from the set of solutions that drive branches that have not been fully executed (B_{NFE}).

Calculating CurrFitness From Fitness. For the Branch Coverage fitness functions as a new path is discovered the fitness will increase significantly; to balance this the fitness used by the simulated annealing algorithm is taken to be the average of the previous fifty results. A moving average is used in order to ensure that the algorithm continues to investigate newly discovered areas of the search space, by spreading out the fitness spikes seen at this point over the next set of iterations.

3) Maximum Loop Counts: **Loops (Lo)** calculates the average number of iterations of each loop on the current path, the result is then normalised using the maximum observed number of iterations. The algorithm is based on previous work by Khan [22]. Using the CFG in Figure 1; block H will be identified as a loop back edge, the fitness for the solution in this case will be calculated as the number of times block H has executed on the current path. In cases where there is more than one loop then the average number of iterations for all loops in the test item will be calculated as the fitness. As a final step the fitness is normalised by dividing the fitness by the highest fitness ever observed. Equation (5) shows the operation of the fitness function, where L_P represents the number of iterations for each loop on the current path, and N_L the number of loops on the current path.

$$CurrFitness_{Lo} = \frac{1}{Fitness^{max}}\frac{1}{L_p}\sum_{l=0}^{L_p}(LoopIter(l)) \quad (5)$$

Branch Coverage Loops (BCHLr) aims to target one of the issues identified with the BCH fitness function, in that the function has a poor focus on maximising loop counts. The function combines the result produced using BCH, with the result using Lo to produce a fitness function that begins by trying to identify unseen blocks, but evolves as the search progresses to concentrate on identifying higher loop counts. Equation (6) illustrates how the fitness is calculated; W_L is used to weight the effect of the loop fitness calculation (Lo) and is initialised to zero.

$$CurrFitness_{BCHLr} = \frac{(W_L * CurrFitness_{Lo}) + CurrFitness_{BCH}}{1 + W_L} \quad (6)$$

As the test progresses, and the branch coverage obtained increases then W_L, the loop fitness weighting, is increased. This changes the priority of the fitness function as the test progresses to focus on maximising loop counts.

4) Changes in ET: **Unique Execution Times (UET)** as an indication that a new path has been traversed. Paths themselves are not monitored as maintaining a list of which paths has been executed and then checking against this list would be too slow. The fitness function keeps a record of each solution's execution time, and counts how many times each unique time has been observed, the fewer times the execution time of the current solution has been observed, the better the fitness of the solution. This is defined by Equation (7) where $TimeCounter$ is an array that stores a counter for each execution time value, so a newly observed execution time would return a $TimeCounter$ value of zero.

$$Fitness_{UET} = \frac{1 - TimeCounter(CurrTime)}{100} \quad (7)$$

The algorithm is designed as a simple path coverage metric, and is designed to provide a wide execution of the solution space. As the same execution time is observed its fitness will slowly decrease. This ensures that the space around previously observed execution times is still explored.

IV. METHOD

In this section we describe the method behind an experiment conducted in order to test the effectiveness of each of the defined fitness functions.

The analysis makes no assumptions about, and no restrictions have been placed upon the code under test. The industrial code used has been designed in SPARK-Ada against DO-178C (ED-12C) standards, although this was merely a consequence of the available industrial code and does not represent a restriction on the method used. To show that this is true, other code examples from WCET benchmarks have also been evaluated.

A. Test Code Items

Twelve code items were used to test the effectiveness of each fitness function. These code items include four standard benchmarks as well as eight industrial test code items.

The standard benchmarks used for the analysis were taken from the Mälardalen WCET Benchmarks [8] and the TACLeBench collection of benchmarks [25]. A large number of the benchmarks were not included as they provided constant execution times when executed on the target processor and hence were not sufficiently interesting. The benchmarks used were chosen as the execution time of each varies significantly as the input search space is traversed, and because they contain input data dependant loops.

The industrial test code used for the analysis was taken from a Rolls-Royce engine control system and has been designed and verified according to DO-178C (ED-12C) standards as a level A package [6]. The items chosen were selected as they represent a broad cross section of the engine control system software, and provide a real life example of industrial software. Some items contain complex constructs, input dependant loops or infeasible paths, whereas other items are more simplistic and contain fewer branches and simpler constructs. This is important as any automatic analysis must analyse simple functions efficiently and recognise when to stop the analysis.

Each code item is first instrumented by the RapiTime tool [4]. Instrumentation Points, or iPoints, 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. Table I summarises each of the test code items used for the analysis. The table defines whether each item contains any loops (L), the number of executable lines of code of each item (LOC). The McCabe Cyclomatic Complexity (MCC) [26] for each item, including all called functions is also listed, this provides an indication of the minimum bound on the number of paths through the program. The number of inputs for each item is shown, these are broken down as I/F/B/S - Integers/Floats/Booleans/States. As the table illustrates a wide selection of software components, of varying complexity, were chosen for the analysis. The States flag illustrates the number of 'state variables' that are carried forward to the execution of the test code item from its previous execution. The Rolls-Royce items of the analysis are taken from a control system which incorporates a large amount of feedback, this means different test iterations are influenced by previous system state, setup by previous test iterations. This is an important feature that cannot be ignored, and so as each test executes the state from previous test iterations is carried forward 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.

TABLE I
TEST CODE ITEMS USED FOR THE ANALYSIS

Name	Source	L	LOC	MCC	Inputs I/F/B/S
QSort	MB	Y	121	21	0/20/0/0
Qurt	MB	Y	166	19	0/3/0/0
Select	MB	Y	114	20	1/100/0/0
InsertSort	MB	Y	7	5	100/0/0/0
F	Rolls-Royce	Y	1101	154	0/17/12/250
ACDF	Rolls-Royce	N	85	9	0/7/4/22
ACDN	Rolls-Royce	N	167	14	0/6/6/25
ACDP	Rolls-Royce	Y	254	27	0/8/5/22
ACDT	Rolls-Royce	Y	395	55	0/26/13/66
VCA	Rolls-Royce	Y	590	68	1/40/17/21
VCP	Rolls-Royce	Y	922	94	1/44/43/30
VCS	Rolls-Royce	N	205	21	0/6/2/0

There is an argument that each state variable contained within each test code item should be modelled as an input, however in this experiment only the inputs at the root function of the analysis were controlled. This is because the analysis aimed to identify how effective the algorithm could be based on minimal input from verification engineers. So functions lower down in the call tree are only controlled from the highest level.

B. Experiment Setup

The search algorithm was evaluated fifty times for each fitness function on each test code item. Each test was started with a random seed, and a random selection of starting data inputs. All tests were executed in a cycle accurate simulator targeting the Rolls-Royce processor, this was to allow a large number of tests to be executed simultaneously, the software was at all times compiled to target the processor itself.

As each test executes it reports its current observed HWM, and its current iPoint coverage. As each iPoint is encountered the iPoint and a timestamp are output to the file system, and the iPoint ID is written to processor memory for use by the fitness function. Following completion of the test this trace data is fed into the RapiTime tool to provide a RWCET figure for the test.

Finally the fifty fitness function results were analysed using a statistical test [27] [28]. A p-value of less than 0.05 was obtained which showed that the results were scientifically and statistically significant, in other words there was a clear trend as to which approach was better and that this was not due to random chance. Therefore it could be concluded fifty tests was sufficient.

C. The Rolls-Royce Processor

The Rolls-Royce Processor is a packaged device that integrates a core, memory, IO and tracepoint 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 features a five-stage superscaler pipeline, with multiple execution units allowing managed parallel execution. The processor also implements simple static branch prediction logic. The processor does not incorporate a data or instruction cache.

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.

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.

The trace facility is an independent component within the processor, separate to all peripherals. The output of iPoints is performed on a reserved interface, thus allowing iPoints to be safely, non-intrusively, kept in the final software verified and delivered with no disturbance on data buses. This ensures that the final code delivered to customer is identical to the code analysed and verified [5].

V. Experiment Results

This section presents the results obtained following the experiments described in section IV.

A. Objective 1 - Efficiency

It could be argued that a well designed search algorithm left to execute indefinitely would provide perfect results. However even if the execution is automatic, the use of test hardware is still costly, and in a large industrial project the number of tests that can be executed runs into the thousands, as they need to be executed in time to meet project deadlines. Therefore efficiency is a key requirement for any analysis tool targeted at industry.

One aim of this analysis was to produce reliable RWCET estimates, within a bounded time-frame. To assess the efficiency of each fitness function the HWM for each test iteration was collected during test execution. The mean HWM for each fitness function, at each iteration was then calculated and plotted for analysis. For the majority of the test code items the test results for each fitness function varied by less than 10% as each test progressed, however in the cases of ACDT, VCP and VCA the difference was more profound, this is illustrated in Figures 2 and 3.

Firstly all individual tests for all fitness functions on all test code items completed in less than 20,000 test iterations, this took approximately twelve hours to execute in simulation. In the case of the simple test code items each test completed in approximately 2000 iterations, which took on average one hour to execute.

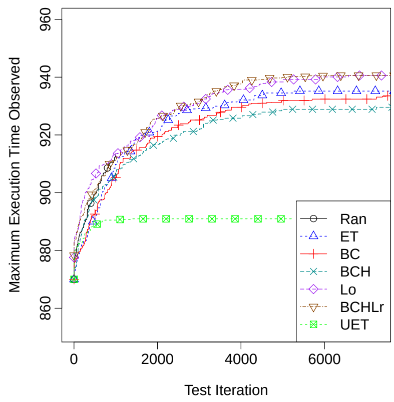

Fig. 2. ACDT Mean HWM Observed as the Test Progresses

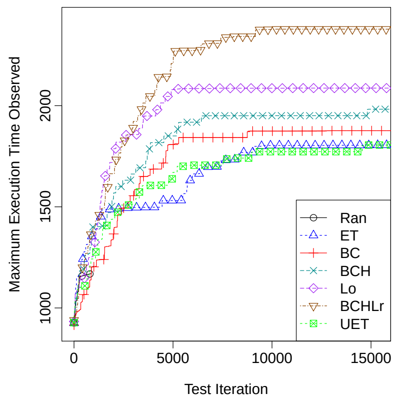

Fig. 3. VCA Mean HWM Observed as the Test Progresses

In an industrial context if each test takes one hour, provided there was enough server power to allow multiple concurrent tests, this could be deemed as acceptable. However for the more complex functions the fact that each individual test takes twelve hours illustrates the importance of identifying a test result efficiently. It also illustrates how it is important for the algorithm to identify when no more progress is being made, and to stop searching - this is particularly pertinent for the small functions which may not see a great deal of improvement across their test run.

Figure 2 shows the mean HWM for the ACDT test item, over time for each fitness function, which provides a representation of test progression. The graph shows how as each test progresses all the fitness functions were able to obtain results similar to each other, with the exception of UET. One possible reason for this is the size of the input space, and number of complex paths through this function, that the UET fitness function was not able to manipulate as effectively.

VCA, shown in Figure 3, on the other hand presented a much larger difference in mean HWM figures, in this case BCHLr was able to produce the best observed HWMs and

TABLE II
OBJECTIVE 2 - THE NUMBER OF TESTS THAT ACHIEVED GREATER THAN 90% IPOINT COVERAGE

Item	MCC	Ran	ET	BC	BCH	Lo	BCHLr	UET
Qsort	21	50	50	50	50	50	50	50
Qurt	19	46	48	49	43	48	48	49
Select	20	50	50	50	50	50	50	50
InsertSort	5	50	50	50	50	50	50	50
F	154	50	50	50	50	50	50	50
ACDF	9	47	48	50	49	49	50	38
ACDN	14	38	48	45	46	48	49	34
ACDT	55	50	50	49	50	50	50	42
ACDP	27	29	25	45	43	40	45	14
VCA	68	6	25	28	30	32	42	24
VCP	94	13	25	28	25	32	35	30
VCS	21	50	50	50	50	50	50	50
Mean		40	43	45	45	46	47	40

largely leads throughout the test. By 10,000 iterations all the fitness functions had stopped improving.

In summary the progression of each fitness function's progression over time illustrated that all the algorithms were capable of producing results efficiently for the simple code functions, for the more complex functions BCHLr performed well over all functions, with Lo, ET and BCH able to produce good results in most of the test code items.

B. *Objective 2 - Reliable Coverage*

Industry cannot rely on just reliably achieving a high predicted RWCET as for certification it is important we are able to argue about confidence in the degree of coverage. The objective of this section is to evaluate the relative coverage achieved by the different approaches by reviewing the iPoint coverage during each test.

Table II shows the number of test runs for each fitness function that obtained iPoint coverage within 90% of the maximum possible.

For all of the simpler test code items, those with McCabe complexity of 21 or less the iPoint coverage for all fitness functions was 100% in most cases. The other tests showed lower iPoint coverage for some of the fitness functions. Again this showed for simple code items all of the fitness functions were able to obtain reliable results.

For the more complex functions the variance between fitness functions was more profound. A number of the functions, such as ACDP, F and VCP contain a number of hard to reach paths. For ACDP for instance the branch coverage fitness functions were able to narrow in on hard these reach paths more reliably, and thus achieved higher iPoint coverage. The tests that achieved higher iPoint coverage, for instance BCHLr and BCH also obtained higher RWCET figures later on.

The VCA test code item showed a huge variance in iPoint coverage, as can be seen in Figure 4. The function features a large portion of difficult to reach control code, this code accounts for roughly 40% of the iPoints inside the function, as the results showed. Only BCH, Lo and BCHLr were able to reliably traverse this hard to reach path, and obtained consistently high iPoint coverage.

In the case of VCP, Figure 5, again ET and BC failed to obtain consistent branch coverage. One contributing factor to this was the size of the input space for VCP, which is considerably larger than a number of the other test code items, and results in a much larger search space.

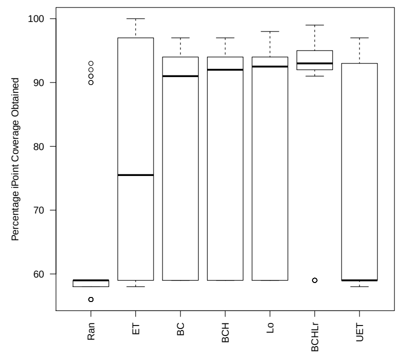

Fig. 4. iPoint Coverage Obtained for the VCA Code Item

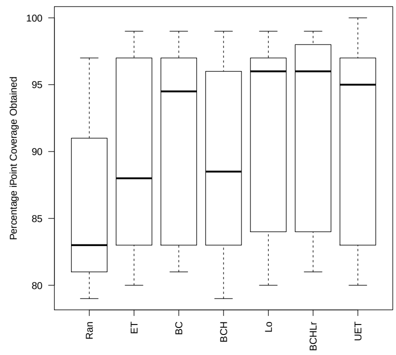

Fig. 5. iPoint Coverage Obtained for the VCP Code Item

Throughout this paper the boxplot whiskers are set to 5

In summary BCHLr has again been shown to provide reliable results across all test code items. Other fitness functions, such as BCH or Lo, were able to similar results, but also produced poorer results in other test code items, such as VCA and VCP. This was shown to be because BCHLr was able to execute specific hard to reach paths, without a focus on reaching these paths, other fitness functions, like ET were unable to reliably achieve high iPoint coverage, and consequentially achieved poorer RWCET figures.

C. *Objective 3 - Reliable RWCET Results*

This objective is analysed by reviewing the results produced by the RapiTime tool, the RWCET. As the AWCET is not known each individual test is executed for significantly longer than necessary, and the results between all tests were then compared against each other. This allows an assessment to be performed into the reliability of each individual fitness function, with particular attention paid to the results when compared to the ET fitness function. A comparison between the maximum HWM obtained and the RWCET calculated is performed to assess how the data input guides the result. The

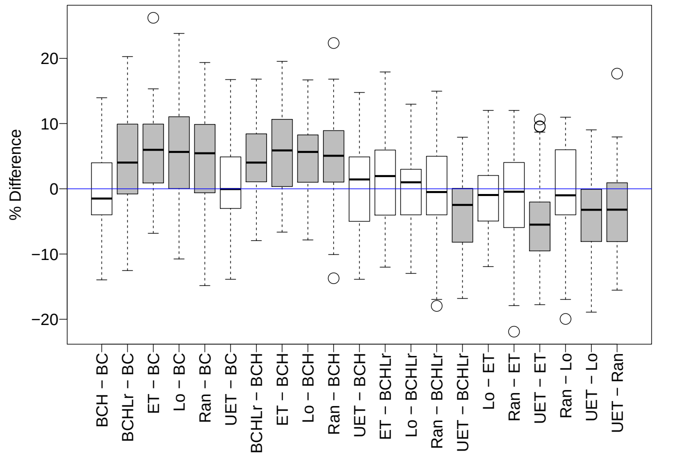

Fig. 6. Comparison of the distribution differences for each fitness function combination, for InsertSort. Shaded plot indicates significance.

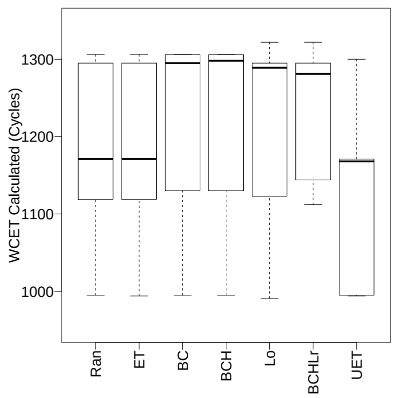

Fig. 7. RWCET Results Calculated for the ACDP Test Item

median is used throughout this evaluation as it best reflects where the majority of the results lie, this reflects the aim of this analysis - to produce good approximations the majority of the time, rather than a better result only once.

Finally a statistical analysis was used to assess whether any of the RWCET distributions from each fitness function was significantly different from any other, this was used in order to confirm the results represented a large enough sample to show significance [27]. The data analysed is non-parametric (does not follow a normal distribution) and only one data source was used therefore a Friedman test with an alpha level of 0.05 was chosen for the analysis. This revealed that there was a significant difference between the fitness functions for all tests, this is denoted in this section as the Friedman chi-squared result (χ_r^2), the degrees freedom and the p value. Following the Friedman test a Wilcoxon-Nemenyi-McDonald-Thompson [28] was used to reveal which fitness functions produced significantly different distributions, the results being displayed using boxplots.

For the smaller code items, with McCabe complexities of 21 or less, the variance between each fitness function was very low; all fitness functions obtained RWCET figures within 10% of each other, with ET generally performing best. For the InsertSort test code item the overall Friedman test result was $\chi_r^2(6) = 67.8, p{<}0.01$ indicated an overall significance. Figure 6 illustrates the results of the Wilcoxon-Nemenyi-McDonald-Thompson test; a shaded boxplot indicates a significant result (p <0.05). The figure shows how the Lo, ET and BCHLr were able to achieve consistently better results than the fitness functions that just focused on iPoint coverage alone - this was most likely because the InsertSort test code item only contains 5 iPoints, which are fully executed very early in the search.

For the Mälardalen WCET benchmark functions there was a large difference between the HWM and RWCET, Qsort for instance observed a HWM for each fitness function of between 8% and 15% of the calculated RWCET. In a similar vein the difference between fitness function's was marginal for the Select code item, however the HWM observed was less than 1% of the RWCET calculated. This was because of the effect of an infeasible path which spans over a triple depth nested loop.

The largest difference between the different fitness function's RWCET results was produced by the VCA, ACDP and VCP code items.

The VCP code item, shown in Figures 8 and 9, exhibited a significant variance of up to 30% between RWCET figures ($\chi_r^2(6) = 84.9, p < 0.01$). This was found to be due to the size of the input space which led to a significantly larger search space and a lower resultant iPoint coverage. As well as this the function contains a number of loops whose execution is reliant on the data input into the test code. This highlighted one flaw with the BCH/BC fitness functions, in that they were unable to focus the algorithm on increasing the number of iterations of the loops found in the test code. The loop coverage fitness functions, in particular BCHLr, were able to exploit this type of code construct, and produced the most consistent, highest result.

The VCA function showed a strong correlation between iPoint coverage (Figure 4) and RWCET (Figure 11). For VCA the variance between the maximum and minimum RWCET results approached 50%, as shown by Figures 10 and 11 ($\chi_r^2(6) = 91.5, p < 0.01$). The median RWCET figures obtained by BCH, BCHLr and Lo were almost three times as high as the median RWCET obtained by other fitness functions. Investigation revealed the reason for the difference in RWCET was due to the fact that the worst case path in these software functions was along a hard to reach path, so only the algorithms able to reliably traverse this path identified the highest RWCET.

VI. Conclusions

MBTA reduces the cost of obtaining reliable WCET figures, however the current techniques available are only as reliable as the data that is fed to them. Where the right answer is not necessarily available, a reliable and robust process for identifying a good WCET figure is essential.

Our work presents fitness functions devised to target MBTA. In particular one fitness function that targets branch coverage, and loop counts, has been shown to produce better WCET

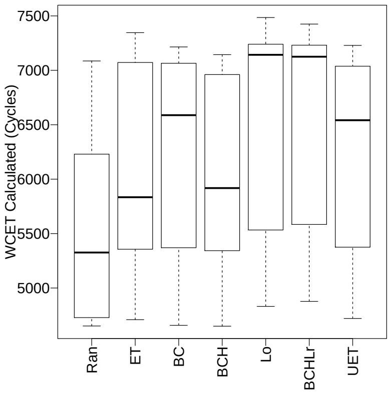

Fig. 8. RWCET Results Calculated for the VCP Test Item

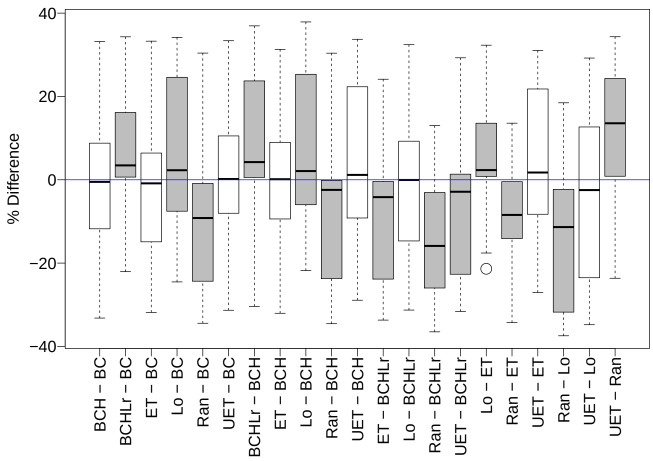

Fig. 9. Comparison of the distribution differences for each fitness function combination, for VCP. Shaded plot indicates significance.

figures than fitness functions that target larger execution times alone. This was found to be because the fitness function was better able to focus on blocks of code that have not been traversed, or are found on difficult, hard to reach paths.

The analysis targeted a real industrial deterministic processor used for executing safety critical software, as such elements of processor state could be negated. Were the analysis to target a less deterministic processor then the fitness functions could be altered to add additional WCET increasing features, such as maximising cache hits, or branch misses.

On the whole the simulated annealing algorithm and fitness functions produced reliable coverage across all test code items, possible options for increasing the reliability of the approach could be to look at simplifying the search space, for instance by identifying key variables that affect the WCET [29], or indeed using the compiler for assistance [30].

In summary for the simple code items all the fitness functions were able to obtain results similar to each other. As the code items complexity grew BCHLr was able to obtain more consistent results across all the test code items. This was because BCHLr was able to focus on branches not properly exercised before and easily re-target the search on areas of poor coverage, this helped the function traverse a number of

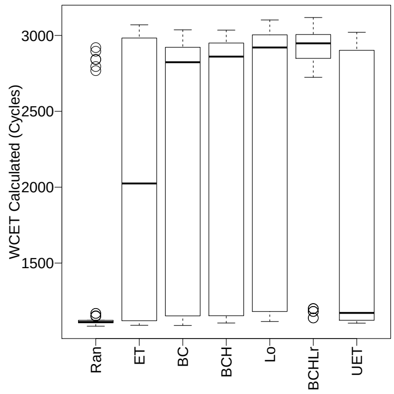

Fig. 10. RWCET Results Calculated for the VCA Code Item

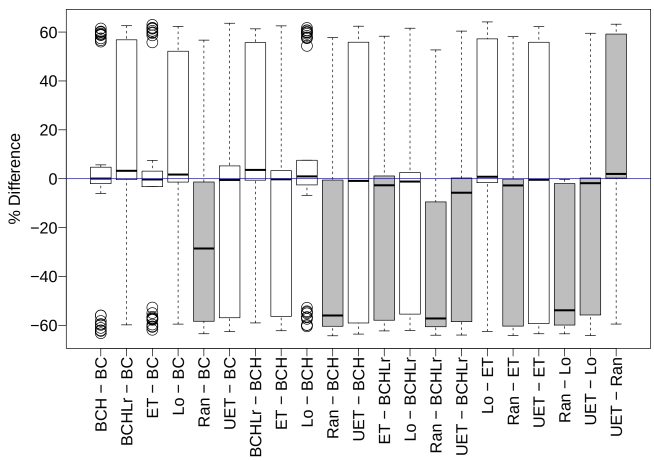

Fig. 11. Comparison of the distribution differences for each fitness function combination, for VCA. Shaded plot indicates significance.

difficult to reach paths that other functions were unable to reach. As the search continued though the algorithm was then able to increase the loop counts observed during testing to better boost the results.

VII. Acknowledgments

We would like to thank Frank Soboczenski, Benjamin Lesage and Mike Bennett for their valuable contributions and comments.

References

[1] Y.-T. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in *ACM SIGPLAN Notices*, vol. 30, no. 11. ACM, 1995, pp. 88–98.

[2] R. Kirner and P. Puschner, “Obstacles in worst-case execution time analysis,” in *Proceedings of the 11th International Symposium on Object Oriented Real-Time Distributed Computing*. IEEE, 2008, pp. 333–339.

[3] N. Tracey, J. Clark, K. Mander, and J. McDermid, “An automated framework for structural test-data generation,” in *Proceedings of the 13th International Conference on Automated Software Engineering*. IEEE, 1998, pp. 285–288.

[4] Rapitime explained: White paper. [Online]. Available: http://www.rapitasystems.com/downloads/brochures-white-papers/rapitime-explained

[5] S. Law, M. Bennett, I. Ellis, S. Hutchesson, G. Bernat, A. Colin, and A. Coombes, “Effective worst-case execution time analysis of DO178C level A software,” *Ada User Journal*, vol. 36, no. 3, pp. 182–186, 2015.

[6] RTCA, "DO-178C - Software Considerations in Airborne Systems and Equipment Certification," 2011.

[7] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "WCET coverage for pipelines," *Real-Time Systems Research Group - University of York and Institute of Computer Engineering - Vienna University of Technology, Technical Report*, 2006.

[8] C. Ballabriga, H. Cassé, and M. De Michiel, "The Mälardalen WCET benchmarks: Past, present and future," in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, 2010.

[9] J.-F. Deverge and I. Puaut, "Safe measurement-based WCET estimation," in *Proceedings of the 5th International Workshop on WCET Analysis*, vol. 5, 2005.

[10] S. Stattelmann and F. Martin, "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*, 2010.

[11] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling," in *Proceedings of the 22nd Real-Time Systems Symposium*, 2001, pp. 215–224.

[12] J. Hansen, S. A. Hissam, and G. A. Moreno, "Statistical-based WCET estimation and validation," in *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*, 2009.

[13] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs," in *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012, pp. 91–101.

[14] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms," *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

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

[16] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 2009, pp. 430–444.

[17] N. Williams, "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*, 2005.

[18] N. Williams and M. Roger, "Test generation strategies to measure worst-case execution time," in *Automation of Software Test, 2009. AST'09. ICSE Workshop on*. IEEE, 2009, pp. 88–96.

[19] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, "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 (ISORC)*, 2011, pp. 144–151.

[20] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement," July 2008.

[21] S. Bünte, M. Zolda, and R. Kirner, "Lets get less optimistic in measurement based timing analysis," in *Proceedings of the 6th International Symposium on Industrial Embedded Systems*.

[22] I. Bate and U. Khan, "WCET analysis of modern processors using multi-criteria optimisation," *Empirical Software Engineering*, vol. 16, no. 1, pp. 5–28, 2011.

[23] S. Kirkpatrick, D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[24] D. Connolly, "General purpose simulated annealing," *Journal of the Operational Research Society*, pp. 495–505, 1992.

[25] "TACLeBench - timing analysis on code level," http://www.tacle.eu/index.php/activities/taclebench, accessed: 2015-09-14.

[26] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[27] S. Siegel and N. Castellan, *Non Parametric Statistics for the Behavioral Sciences*. McGraw-Hill International, 1988.

[28] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods*. Wiley-Interscience, 1999.

[29] J. Zwirchmayr, P. Sotin, A. Bonenfant, D. Claraz, and P. Cuenot, "Identifying relevant parameters to improve WCET analysis," in *14th International Workshop on Worst-Case Execution Time Analysis*, 2014, p. 93.

[30] R. Kirner and M. Zolda, "Compiler support for measurement-based timing analysis," in *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis*, 2011.