Establishing Confidence and Understanding Uncertainty in Real-Time Systems

Iain Bate
University of York
York, United Kingdom
iain.bate@york.ac.uk

David Griffin
University of York
York, United Kingdom
david.griffin@york.ac.uk

Benjamin Lesage
University of York
York, United Kingdom
benjamin.lesage@york.ac.uk

ABSTRACT

A benefit with traditional static analysis approaches to single criticality hard real-time systems is that the uncertainties, and hence confidence, associated with timing requirements being met are better understood than Measurement-Based Timing Analysis (MBTA) approaches. In brief, failures are mostly accounted for by human errors or random hardware failures. With the introduction of measurement-based approaches to timing analysis to help deal with more advanced processors, the situation is much more complex. The complexity comes from new sources of epistemic failures: imperfect timing measurements from the system, approximations in the analysis, and the conscious decision that parts of the system are not always guaranteed to be scheduled in a hard real-time manner. The goal of this paper is to establish some understanding of the uncertainties based on a proposed industrial approach to MBTA and consequently how confidence in the system's timing measures can be managed. More specifically understanding the epistemic uncertainties associated with measures used for timing analysis concentrating on whether it could have been foreseen that: further testing with a given method could have avoided failures; and deficiencies with the current testing method could have been predicted.

ACM Reference Format:
Iain Bate, David Griffin, and Benjamin Lesage. 2020. Establishing Confidence and Understanding Uncertainty in Real-Time Systems. In *28th International Conference on Real-Time Networks and Systems (RTNS 2020), June 9–10, 2020, Paris, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394810.3394816

1 INTRODUCTION

There are a wide variety of models for mixed-criticality scheduling. A pre-requisite for all of these is that reliable execution times are available for the Software Under Test (SUT) and from these values for C_{Lo}, C_{Hi} can be deduced. C_{Lo} is the Worst-Case Execution Time (WCET) of all jobs in normal mode and C_{Hi} may be the WCET for high-criticality jobs after a *functional mode change* has occurred into *High-Criticality Mode (HCM).* As motivated by Graydon [13] any time a different set of tasks are scheduled, a functional mode change is considered to have occurred and the set of tasks that execute at any given time is a functional mode.

To date, all work that we are aware of has not considered how the reliability of execution times can be demonstrated, and how C_{Lo} and C_{Hi} are determined. The general academic assumption is C_{Lo} comes from testing and may be optimistic. C_{Hi} would then come from of pessimistic form of analysis. An industrial perspective from Law [5] is that both C_{Lo} and C_{Hi} could be obtained by the same data, however C_{Hi} could feature paths through the SUT that are only executed under exceptional circumstances, e.g. when there are hardware faults, as well as by the application of pessimistic WCET analysis to the measurements. However the values are determined, the confidence in the input data to the process needs to be understood. The *input data* is the measurements (including execution times) that are observed after a test vector is applied to the SUT. The contributions of this paper are establishing methods for determining:

(1) how reliable the timing measurements are;
(2) the likelihood of the current testing method determining new (significant) information;
(3) if infeasible paths might exist that could be used to optimise hybrid analysis; and
(4) whether there are sufficient measurements to support the above.

The contributions are evaluated using an industrially-proven technique for generating execution time measurements, however our belief is that the contributions are generally applicable. The structure of the paper is as follows. The paper begins with background and related work before providing an in-depth description of the testing method used as a basis for the paper. Section 4 then investigates whether the outputs of testing give a reliable input to MBTA. Then, section 5 explores whether the current testing method is likely to determine new information. Section 6 looks at ways of determining if the software might have infeasible paths based on the available test data. Finally the paper concludes with observations of what can be determined, methods for predicting the observations that could be determined, and the confidence associated with those predictions.

2 BACKGROUND AND RELATED WORK

The related work section is split into the following three parts: research on static analysis to help understand the main influences on the WCET of software; methods for generating data as part of Measurement-Based Timing Analysis; and techniques for arguing the reliability of software.

RTNS 2020, June 9–10, 2020, Paris, France
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7593-1/20/06…$15.00
https://doi.org/10.1145/3394810.3394816

2.1 Static Analysis for WCET

The purpose of this section is not to provide a thorough review of static analysis for WCET, but instead understand the main inputs to the static analysis approaches as these may be considered significant factors affecting the reliability of execution time data. The initial work on WCET analysis concentrated on the different paths through the program [28, 29]. In this work, there were two key influences: the blocks that were executed and the paths subsequently taken. These were dictated by the decisions taken at branch instructions and the number of times around a loop.

The subsequent areas of work analysed a wide variety of processor features including pipelines [24], caches [26] and branch prediction [10]. Whilst all these factors influence the execution times of software, Khan showed that their influence was much less significant than the path through the software [16, 17]. A second reason for not considering these further is the difficulty in obtaining the measurements on a real target, rather than a cycle-accurate simulator, and that taking the measurements is likely to introduce significant measurement noise [22, 23].

2.2 Generation of Execution Time Data

The purpose of this section is to review the approaches to generating execution times to support MBTA. The review is intended to be independent of how MBTA is performed with the key techniques being: to take the High WaterMark (HWM) or in other words the maximum execution time; hybrid analysis where measurements are combined with results from static analysis [6]; or techniques based on probabilistic analysis [9]. It is noted that in [11] a number of concerns from the probabilistic analysis communities were raised that are likely to be relevant to all approaches to generating the execution time data.

The main approaches to generating the test data have been search-based approaches. Wegener [31] and Tracey [30] both illustrate 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 [31] 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 framework introduced is primarily based on search algorithms, which when compared to HWMs observations from system-level testing, produced good results. However the drawback is the tool had to achieve path coverage to obtain a sound WCET and path coverage was not targeted by the search.

Wenzel [32] 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 [32]. 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 [34] 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 [35] with an analysis into possible simplifications that can be made to avoid the analysis requiring full path coverage. These include 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 [8] examined the effectiveness of using model checking [15] 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 [7] 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 [16, 17] 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 include 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 concludes 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.

More recently, the work of Law [19] has used coverage-based metrics to ensure the reliability of HWM and hybrid analysis based on Rapitime, and to better support certification. The work was then extended in [21] to provide a more scalable approach. As this approach has been shown to more reliably obtain the WCET the other approaches discussed in this section, the works fall short of justifying (with evidence) why it is more reliable and whether further testing would provide better results.

2.3 Justifying the Reliability of Software

Most of the work on justifying the reliability of software has been based on Reliability Growth Models (RGM) with the seminal works in this area using Bayesian approaches [27]. The challenge with these approaches are that they are black box in nature which means the principal causes of a lack of reliability are not considered and they assume each fault is independent with respect to the previous ones. Graydon addresses the first of these issues by producing an argument as to how timing is approached for systems that have to be certified [12]. This work concentrated on arguments that could be made without necessarily reviewing the underpinning evidence, i.e. the actual nature of the data that might be collected based on a particular testing approach. Finally there is the previous mentioned work on probabilistic techniques, e.g. [9]. Again this is a black box approach and these techniques concentrate on analysing the data rather than the integrity with which it is captured.

3 THE EXAMPLE TESTING METHOD

The purpose of this section is to introduce the search algorithm used and then the platform on which the software operates. Details of the software are not provided as it is industrial software from an aircraft engine control system, however it has similar characteristics to those in [19] and it is the same examples as used in [21]. For reasons of space only one SUT is presented, however similar trends were observed for the others tested from [21].

3.1 Search-Based Test Case Generation Used

The TACO framework relies on a derivative of a simulated annealing (SA) algorithm [1], outlined in Algorithm 1 for the search. The algorithm starts from a random solution, i.e. a valid test (or input) vector for the function under analysis (line 2). A new solution is generated on each iteration by altering one randomly selected input in the current test vector (l.7). Both operations, generation and mutation, respect the type range constraints of the input vector. The SUT is then executed using the new solution while collecting information on its execution path and timing behaviour (l.8). The new solution is accepted as the new baseline one (l.13) if there is an improvement. This relies on the evaluation of the solution's fitness according to its execution against that of previous solutions. A solution may also be pseudo-randomly accepted (l.12) to ensure the search does not get stuck in a local minima especially in early stages of the search. As the test progresses the pseudo-random selection of poor solutions will decrease, as controlled by the decreasing temperature parameter (l.22). The search stops (l.23) after a minimum number of iterations, if no solutions have been accepted for a few iterations, or the temperature hits a specified lower bound. Additional steps (l.20), such has reheating [18], are taken to prevent the algorithm being caught in a local minima. The key configuration points for the Algorithm and related fitness function are given in Table 1.

3.1.1 *BCHLr Fitness Function.*

The BCHLr fitness function is a coverage-based heuristic to evaluate the fitness of a solution against it predecessors during the search. The fitness of a solution combines three factors:

- Branch Coverage (BC) - Accept solutions which cover new branches to increase path coverage 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 count to maximise the number of iterations of each loop through the code, as proposed by Khan [17].

Algorithm 1 Simulated Annealing

```
1  temperature  := INITIAL_TEMPERATURE
2  currSol := RandomSolution()
3  currTemp := temp
4  rejSols := 0
5  do
6    -- Generate and evaluate new solution
7    newSol := Mutate(currSol)
8    newStats := CallFunction(newSol)
9    newFitness := EvaluateFitness(newStats)
10
11   -- Accept or reject solution
12   if AccSol(newFitness, temp, rand(0..1))
13     currSol, currTemp := newSol, temp
14     rejSols := 0
15   else
16     rejSols := rejSols + 1
17
18   -- Update search temperature
19   if rejSols > HISTORY_TEMPERATURE_SIZE:
20     temperature := currTemp
21   else
22     temp = temp x COOLING
23 loop while not StopAlgorithm()
```

Parameter		Configuration
Temperature	Initial temperature	1.0
	Minimum temperature	0.0001
	Temperature cooling	0.9999
History	History temperature	1000
	History input	10
	History delay	1000
Iterations	Minimum iterations	N.A.
	Maximum iterations	50000
Input	Input change rate	±5% of Value Range
Repetition	Window Size	5
	Window error	5%

Table 1: Simulated annealing search parameters

Branch Coverage (BC) computes the average fitness of the branches traversed during the execution of a solution. A branch fitness is the ratio between the number of unexplored edges out of a branch over the number of edges out of the branch:

$$F_{BC}(p) = \frac{1}{|B_p|} \times \sum_{b \in B_p} \left(\frac{|\{e \mid e \in E_b \wedge \neg Cov(e)\}|}{|E_b|} \right) \tag{1}$$

Algorithm 2 Simulated Annealing - Modifications for Branch History

```
-- Accept or reject solution
[...]

-- Save solution against reached,
-- uncovered branches
for branch in Branches(newStats.path):
  if AllEdgesCovered(branch)
    RemoveFrom(history, branch)
  else if branch not in history
    history[branch] := newSol

-- Reset solution to reach
-- uncovered branch
if rejSols > HISTORY_INPUTS_SIZE
  branch := PickRandom(history)
  newSol := history[branch]

-- Update search temperature
[...]
```

Where B_p is the set of branches traversed by execution path p, E_b denotes the set of outgoing edges from branch b, and $Cov(e)$ captures whether edge e was covered by p or by prior iterations of the search.

Between two solutions, one that reaches a branch with yet unseen outgoing edges will be favoured over one that only covers fully-explored branches. A given solution may further have a different fitness based on the history of explored solutions and branches, and it will decrease in fitness as the search progresses.

The Branch History (H) resets the search to solutions reaching branches with unseen outgoing edges. As each branch executed through a solution is analysed, the history keeps track of the solution and it is stored against that branch. If a branch has unexplored outgoing edges, the stored solution can thus be used as a starting point to reach unobserved outgoing edges. The history triggers when a sufficient number of new solutions has been rejected, and a new matching branch is chosen at random. The input vector stored against this branch is then adopted as the current solution. 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. Algorithm 2 outlines the changes to the simulated annealing algorithm, presented in Algorithm 1, to account for the history.

Maximum Loop Counts (Lr) calculates the average fitness of the loops traversed by a solution as the ratio between the number of iterations across all loops on the path and the maximum number of iterations previously encountered. No prior knowledge is assumed on the maximum iterations of a loop, and it is solely based on the maximum observed count. Like for BC, the fitness of the same solution may thus vary during the search. The algorithm is based on previous work by Khan [17]:

$$F_{Lr}(p) = \frac{\sum_{lo \in Loops(p)} (CountIterations(lo, p))}{MaxIterations} \tag{2}$$

Where $Loops(p)$ captures the set of loops covered by execution path p, $CountIterations(lo, p)$ denotes the number of iterations through loop lo on execution path p, and $MaxIterations$ records the maximum number of iterations encountered in a path during the search (initialised to 1). All iterations through lo from successive executions of the loop, as an example if lo is nested in another loop, count towards the same total; $CountIterations(lo, p)$ does not distinguish between different contexts for loop lo.

The BCHLr heuristic combines the two fitness functions BC and Lr to produce a fitness function that begins by trying to identify unseen blocks, but evolves as the search progresses to favour longer paths through higher loop counts. The two metrics are combined using a weighted sum, with weights W_{Lr} and W_{BC} respectively for Lr and BC:

$$Fitness_{BCHLr}(p) = W_{Lr} \times F_{Lr}(p) + W_{BC} \times F_{BC}(p) \tag{3}$$

As the test progresses, and the branch coverage obtained increases, then the loop fitness Lr weighting (W_{Lr}) increases (and W_{BC} accordingly decreases). This change in weights alters the focus of the fitness function as the analysis progresses from discovering new paths to longer ones.

3.1.2 ET Fitness Function. The ET heuristic attempts to maximise the observed execution time during the search. As each new solution is executed, the execution time of the analysed item is collected as part of the execution statistics, *newStats* in Algorithm 1. This execution time is compared to that of the current solution, the last accepted solution, such that any increase in the current execution time will result in the acceptance of the new solution. Only strict improvements are considered; an identical execution time does not guarantee the new solution is accepted.

$$Fitness_{ET} = \frac{ET(newSol) - ET(currSol) - 1}{ET(Time)} \tag{4}$$

Where $ET(S)$ is the execution time of solution S explored by the simulated annealing outlined in Algorithm 1.

3.2 Platform Configuration Used

In [19], software tasks were used from a Rolls-Royce aircraft engine controller running on a deterministic processor. In this paper the same search algorithm and fitness function is used, however this time a Raspberry PI 3B+ is used as the processor platform as the availability of the deterministic processor was limited. The SUT executes non-preemptively on a single core. Steps are not executing user-level tasks. The Raspberry PI 3B+ have been configured using Linux in such a way that measurement noise is reduced. This is preferred over versions of Linux with the real-time preemptive patches as trials showed the measurement noise was reduced. It is noted Linux has been used over real-time versions of Linux by others [2]. Our detailed assessments did not draw a definitive solution, however our intuition is the main difference of the real-time versions over the *conventional* versions is that fully-preemptive kernels increase the number of possible preemptions, and hence increase the likelihood that the executing user task is preempted. For any form of WCET analysis, including that in this paper, executing atomically (i.e. without preemption) is a fundamental assumption [33].

An illustration of a typical execution profile from a run is shown in Figure 1. The figure illustrates that: the profile is multi-modal in nature, i.e. there are a number of significant distinct peaks; and the range of execution times is approximately the same as the minimum execution time, i.e. the HWM is approximately double the Low WaterMark.

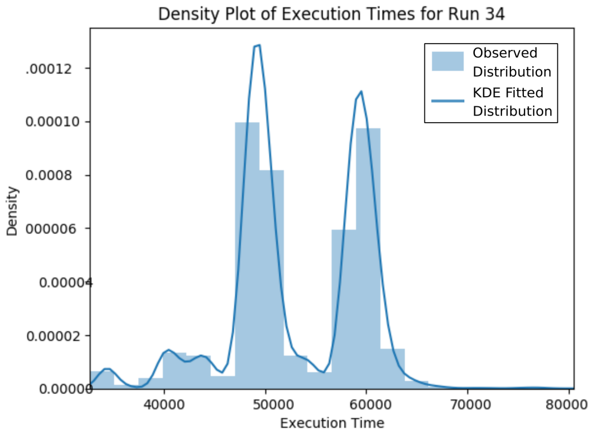

Figure 1: Density Plot Illustrating a Typical Execution Time Profile

4 ENSURING THE RELIABILITY OF TIMING MEASUREMENTS

In this paper, a timing measurement is considered reliable if the same test vector is applied to the SUT a number of times then the variability is bounded and acceptable. Bounding the variability means that subsequent analysis can compensate for it as well as the impact of confounding factors are understood. Confounding factors are those that make the comparison of groups difficult. In WCET analysis terms this means if we say test vector X leads to a longer execution time than test vector Y, then there is no other significant factor (e.g. another software task contending for a shared resource that the SUT is accessing) than the test vector that would lead to the hypothesis being refuted.

The usual confounding factors for timing measures are uncontrolled variables which can be the state of the SUT or the state of the processor. It is noted here that timing measurements are normally performed with the cache flushed so this should not be factor, however the methods presented would include the effects of imperfect cache flushing. The approach advocated in this paper is not an unusual one. Each iteration is repeatedly executed a number of times and the variance in the execution time studied. It is noted that across the repeated executions a check is made that the same path is taken. This was repeated 100 times with each of these trials being for a different path.

Figure 2 presents an example of one set of results represented as a density plot. The y-axis is the frequency and the x-axis the standard deviation, σ_i, of t_i where t_i is the set of execution times for iteration i. This clearly shows a variance in the approximate range of 48,000 to 52,000 which is a range of less than 10%. Next a statistical analysis was performed across all 100 trials. Figure 3 presents another density plot, however this time across the set of 100 trials where the x-axis is the Coefficient of Variation. This figures shows that the vast majority of trials have a standard deviation of less than 5% with respect to their mean. It does show a very small number of trials are higher.

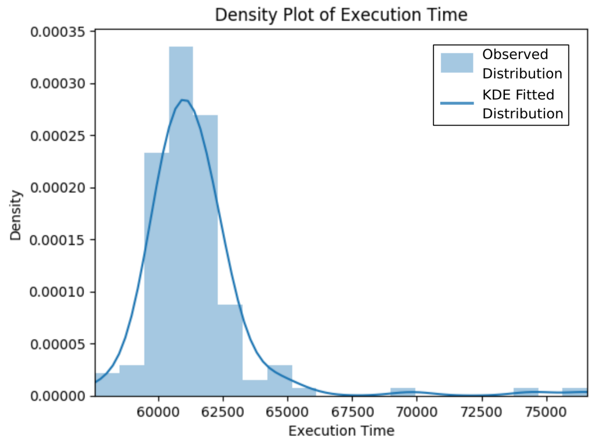

Figure 2: Measurement Noise for an Individual Trial

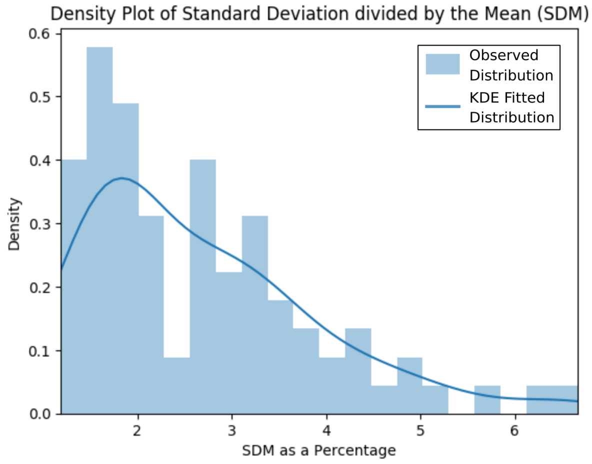

Figure 3: Measurement Noise Across 100 Trials

In summary, the trials presented in this section show that both the execution path and execution time are classed as repeatable. In the case of the execution time, the variance is typically less than 5%. Based on previous experience, 5% was considered an acceptable level.

5 ESTABLISHING THAT THE CURRENT TESTING METHOD IS UNLIKELY TO DETERMINE NEW INFORMATION

The aim of this paper is not to claim the testing method presented is the best approach, instead the aims are to show confidence in the approach and convergence. Confidence is based on the coverage across significant factors which are based on those highlighted in section 3.1. This is considered in section 5.2. This section will consider convergence which is whether the testing method is likely to determine further significant information.

A simple option is just to consider the maximum execution times (or any other measurement) and whether an increased (significant) value has been found in recent history. There are three key issues with this approach. Firstly the maximum value might not be changing but the general distribution of values may be. Secondly a value such as execution time is only one indicator of whether further information is being learned. For example in Law [19] other parameters were used in the fitness function of the search algorithm as they helped guide the search more reliably. The other parameters, e.g. block coverage and loop counts, are also significant factors affecting the execution times of the SUT. Therefore the distribution of these factors is important. Finally semantic understanding of the significant factors may suggest that more significant values for the execution times may be found, e.g. a single iteration hasn't maximised all the loops at the same time which raises the possibility a single test vector could do this. Each of these three issues are considered in turn in the following sub-sections.

5.1 Convergence of WCET

There are two stages to judging whether convergence has occurred. The first is to look at the data and make a subjective assessment of whether the results are changing. The second stage uses statistical analysis techniques to provide a more quantitative assessment of convergence. These are explored in the following sub-sections.

5.1.1 Would the Search Algorithm Performing More Iterations Improve the Results? The first step in the examination of the data is whether the execution time of the SUT is changing. This can be in two dimensions, between iterations and runs. Throughout section 5.1, the experimental approach taken is to randomly take X% of the data (either iterations or runs) and compare it with a different X% of the data. To be specific, if X is runs, then the number of iterations will be fixed so that the impact of a single variable changing (i.e. runs is understood). The process was then repeated with the number of iterations varied and the number of runs fixed. Each trial was repeated 20 times for each value of X.

Examining the number of iterations used provides an argument that running the same test for longer does not have an effect. Figure 4 presents how the average HWM changes (y-axis) over a number of iterations (x-axis), where each iteration is for 100 runs. In this case, after 45,000 iterations the rate of change in the average HWM is insignificant.

Testing the number of runs determines the effect of different starting conditions on the results of the testing method and whether these effects have converged. More specifically if multiple runs are performed with each run having a different starting position, then how do the results change and when do those changes become less significant. Figure 5 presents how the average HWM changes (y-axis) over a number of runs (x-axis), where each run has 10,000 iterations. The results clearly show that as the number of runs increases the average HWM becomes less variable. After 25 runs the average HWM starts to converge, and achieves a consistent value. However, it is also worth noting that all the runs are within 5% which is within the previously established level of measurement noise.

Overall, the results suggest that the rate of change in observations above 50,000 iterations is slow enough that resources would be better spent performing additional runs of the algorithm. These results also suggest that at 10,000 iterations, 25 runs are sufficient, although even at 10 runs it is possible to be within the acceptable 5% error bound. However, using a smaller number of iterations or runs may cause a mis-assessment of the confidence of the result, especially when evaluating a low-probability such as exceeding the observed HWM. Hence the rest of section 5.1 focuses on performing a more detailed analysis.

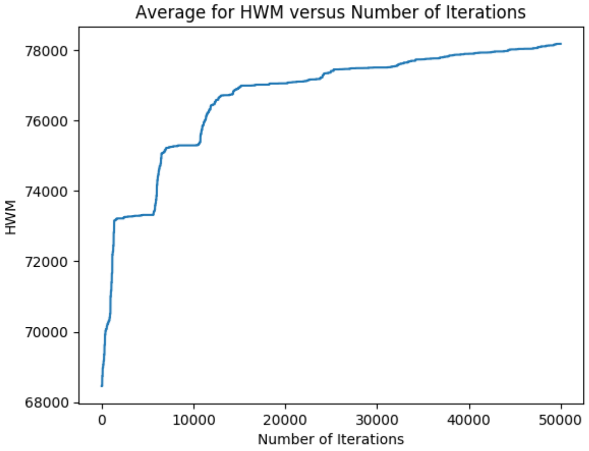

Figure 4: HWM across iterations

5.1.2 Statistical Assessment of Convergence. There are two approaches to examining the convergence: whether the distributions are similar, and how different the distributions are. To assess similarity the Kolmogorov-Smirnov (KS) test is an often applied approach as it does not make assumptions about the nature of the data [25]. A Goodness of Fit (GoF), also referred to as the p-value, of 0.05 is judged as the two samples are drawn from the same distribution. The Earth Movers Distance (EMD) test is often used to assess differences. Therefore in this paper both of these will be used. For the EMD metric, there is no accepted definition of significance threshold. Instead a judgement is made when the distance is not changing as the size of the initial training set increases.

Given appropriate tests of similarity and differences, the experimental approach is to perform cross-fold validation, i.e. for a number of times compare different percentages of an original test set with a revised test set with more results. It is noted the revised test set does not include results from the original test set.

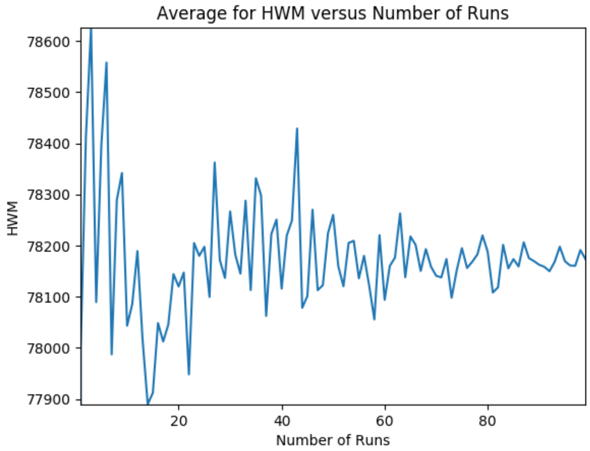

Figure 5: HWM across iterations

For example, the first 10% of the available iterations are compared with different percentages of iterations within a run.

Figures 6 and 7 show the results for the EMD's test and the p-value from the ks-test respectively. The Figures clearly show that both of these metrics stabilise, i.e. do not change by as much, after about 50% of the runs. However, Figure 7 shows that as the number of runs increases, we increase the probability that the experiments are determined to be drawn from the same distribution. The figures show with less data the lack of convergence is more. Even with 50% of the data, there is a significant probability that the KS-test will state that the results are drawn from different distributions and thus the experiment has not yet converged. Taken in conjunction with Figure 6 showing that the distributions of results become more similar the more data is used, this affect is most likely due to extreme results being more likely to be observed the more data is used and causing the KS-test to determine the distributions are different.

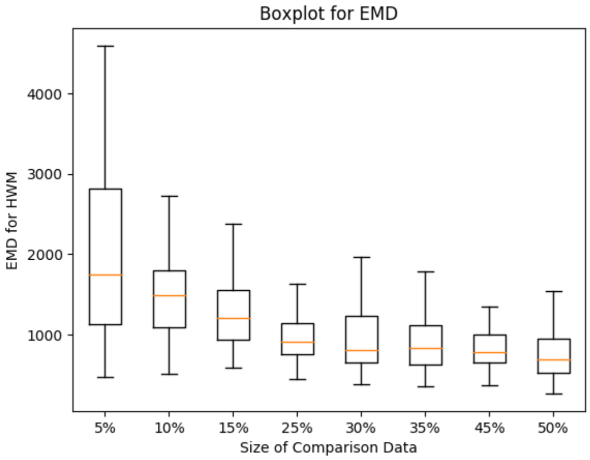

Figure 6: EMD for HWM

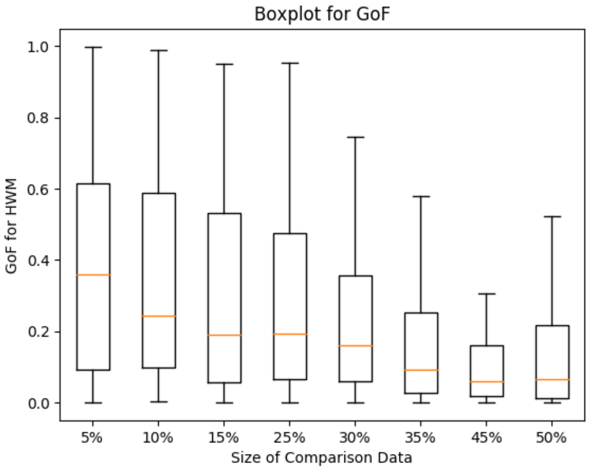

Figure 7: GoF for HWM

In summary, this section has shown conventional methods suggest convergence of WCET has not been achieved despite a significant amount of testing. Therefore, the next section explores whether other more detailed metrics confirm or refute this.

5.2 Convergence of the Significant Factors

In this section, a similar approach to section 5.1 is taken of examining similarities and differences as the number of iterations changes with a fixed number of runs and then a changing number of runs with a fixed number of iterations. The EMD metric is used on the significant factors instead of the p-value from the ks-test as the p-value tended to be zero. The only exception is the HWM as shown in Figure 7. A zero value indicates that the distributions are different, indicating that convergence has not occurred. An alternative reason though is the ks-test is known to be mis-leading when a lot of data is used [20].

Figures are shown for the significant contributing factors: execution times (in Figure 8), path length (in Figure 9), loop bounds (in Figure 10), and iPoints covered (in Figure 11). *Loop bounds* is the number of times each loop in the code is executed in an iteration. *Path length* is the number of iPoints covered in an iteration. An iPoint is an instrumentation point placed at the start of each block in the code in order to record the blocks executed and the times at which each block's execution is started. *iPoints* is the number of different iPoints covered in an iteration. As discussed in section 2.1, other significant factors, e.g. number of cache misses, would be difficult to measure without being obtrusive and are arguably subsumed in the measures already made, e.g. the number of instruction cache misses is affected by the path taken which is related to the path length.

It should be noted that the EMD metric is not comparable across different graphs, due to the underlying statistics being incomparable (for example, loop counts and execution times use different units and thus cannot be compared directly). However, it is possible to compare the trends. In this case, each of Figures 8-10 show a degree of convergence, but still exhibit some variability. This is expected as

BCHLr is a search based algorithm which does not explore the full state-space, and so complete convergence is statistically unlikely. The similarity in patterns suggest that the impact of using more data is similar for each of the significant factors.

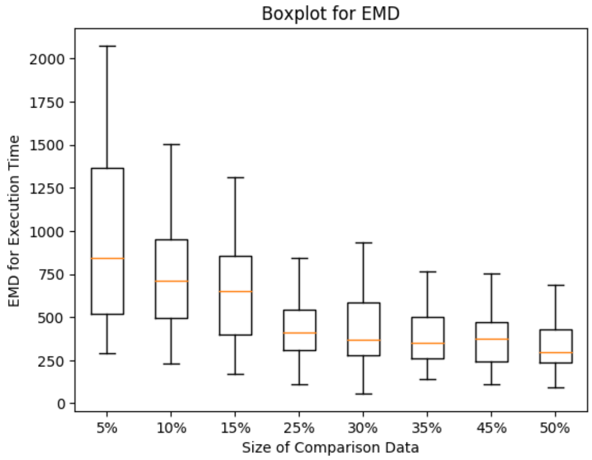

Figure 8: EMD for Execution Times

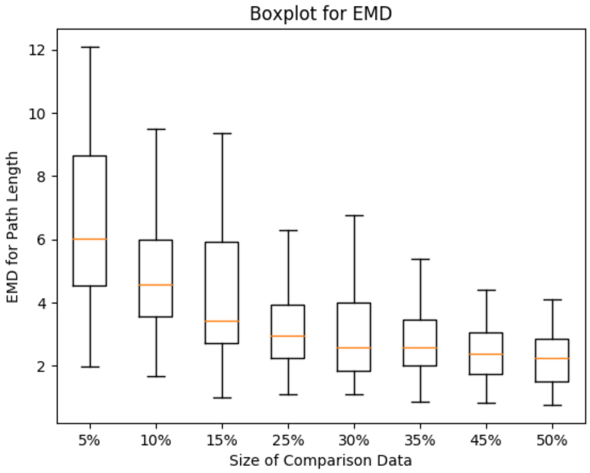

Figure 9: EMD for Path Lengths

Given that the set of measurements seem to have converged, it is now valid to consider the other contributions of the paper.

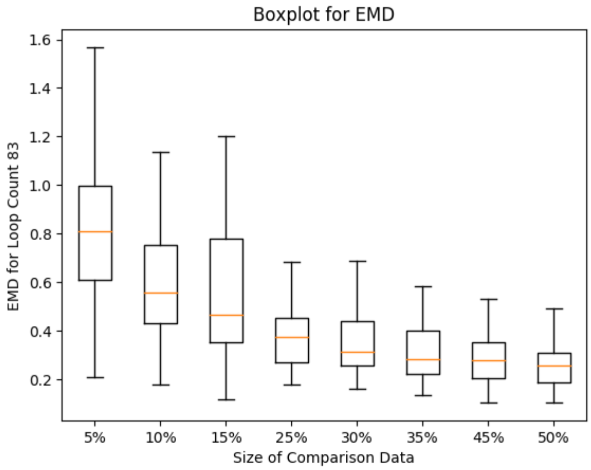

Figure 10: EMD for a Loop Count

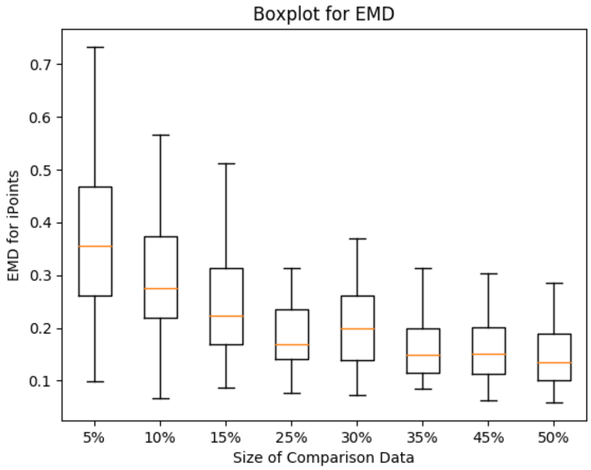

Figure 11: EMD for iPoints

5.3 Semantic Examination of the Results

By casting the hypothesis "The current set of iterations causes the worst-case execution scenario" and trying to refute it, it is possible to argue for or against additional testing. In this section, an approach for trying to refute this hypothesis is presented. The basic approach is to consider the significant factors and consider which of these have been maximised in the same iteration. For example, if the loops in the SUT have maximum observed bounds of L_1, L_2, ··, L_N however not in the same iteration. It is noted previous works, i.e. [3, 4], would allow the maximal loop bounds to be determined given sufficient measurements of the right type. The work in this paper complements this as it helps identify when the current testing approach is unlike to generate new information, i.e. measurements of the right type.

The approach taken in this paper is to consider whether, within a single iteration across all runs of a testing approach all the significant factors are maximised. The significant factors established earlier in section 5.2 are used. That is, loop bounds, path length, and number of iPoints. However, this presented some interesting results: while Path Length was strongly correlated with execution time, Loop Bounds and Maximisation of iPoints were not, despite many observations of these factors being maximised. Further, while these factors were maximised individually with a relatively high frequency, as shown in Figure 12, there were very few observations of these factors being maximised simultaneously. Inspection of the SUT confirmed that this was due to the exact paths that maximised these factors being exceedingly rare within the space containing all paths. This caused the BCHLr algorithm to find these paths and then promptly explore similar paths which did not maximise these properties, which is expected behaviour.

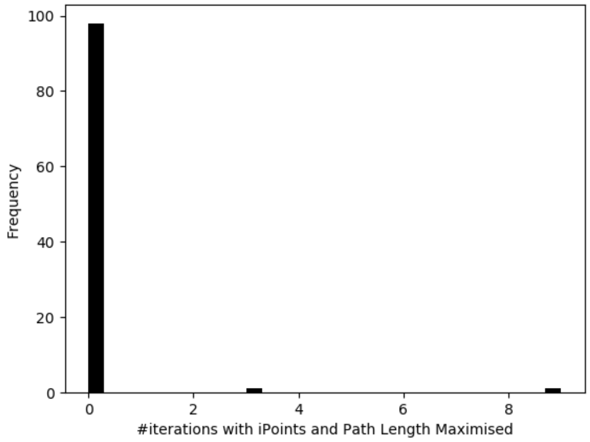

Figure 12: Histogram for iPoints and Path Length Maximised

While the fact that the BCHLr does not conduct extensive testing on paths which maximise the significant factors might cause concern, it should be noted that this is entirely within expectations as BCHLr attempts to maximise coverage rather than execution time. However, such testing should be reserved for factors which do indeed maximise execution time, and of the three factors selected for analysis, only path length correlates strongly with execution time. This can be seen in Figure 13, which shows a clear correlation between Path Length and Execution Time - although some variability is observed, likely due to uncontrolled factors on the Raspberry Pi 3B+ platform. By contrast, while Loop Iterations does not show a high degree of correlation; indeed some of the highest execution times are achieved with some of the lowest loop iterations.

The fact that execution times do not correlate strongly with the number of loop iterations likely comes about due to rarely taken long paths of sequential code (i.e. error handlers) which do not cause a high number of loop iterations. This leads us onto the next section, which investigates how potential infeasible paths can be determined.

6 GUIDING THE USER TOWARDS INFEASIBLE PATHS

Section 4 provides techniques to decide whether the test data is suitable for hybrid analysis. This section looks at how this data can be used to improve the results of hybrid analysis. A key issue with the results of hybrid analysis is the potential pessimism caused by infeasible paths [14]. An infeasible path is defined as a path containing groups of basic blocks that cannot be executed after another group of basic blocks has executed. Previous works to determine infeasible paths by static analysis, e.g. [14] place significant restrictions on developers such as the use of bespoke compilers. The two conditions for infeasible paths are as follows:

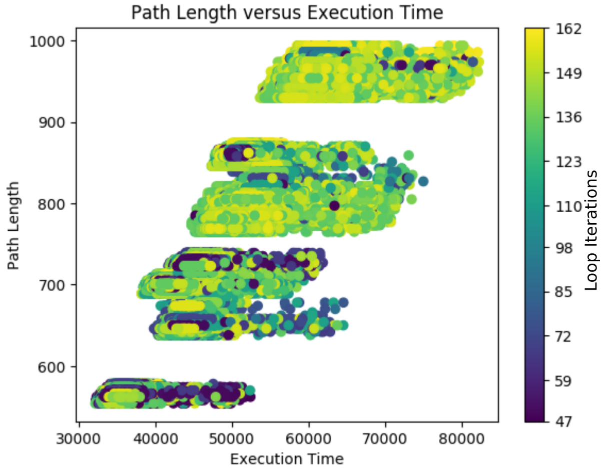

Figure 13: Scatter Plot for the Significant Factors

(1) Two basic blocks never being executed in the same iteration; or
(2) All loop bounds not being maximised in the same iteration.

Algorithm 3 presents a simple example where basic block B would not be executed after basic block A. Algorithm 4 presents a more complex example. The reason the example is more complex is if either X or Y is altered during the execution of basic block A, then both basic block A and B would be executed. Analysis cannot also just rely on inspecting whether basic block A manipulated variables X or Y as, for example, the the code in Algorithm 4 could be preempted and another function could alter the value of X and Y.

Algorithm 3 Simple Example of an Infeasible Path

```
if X == Y then
  basic block A
else
  basic block B
```

Algorithm 4 More Complex Example of an Infeasible Path

```
if X == Y then
  basic block A
if X != Y then
  basic block B
```

As with the detection of outliers, testing and measurement can provide no guarantee of anything. Therefore the approach is again the automatic identification of infeasible paths but providing the human guidance of how to validate the identification. The identification process is described in Algorithm 5, where *NEI* is the set of iPoints not executed in every iteration.

The basis for the identification is to identify sets of pairs of iPoints that are never executed during the same iteration within

Algorithm 5 Identification of Infeasible Paths

```
Determine the set $NEI$
for each iPoint ($i$) in $NEI$
  for all other iPoints (j) in $NEI$
    Check if $i$ and $j$ are ever executed in the same iteration
    If not then store pair of $i$, $j$ in set $NT$
Organise iPoints in set $NT$ into contiguous ranges
Determine frequencies of iPoints in $NT$
Determine execution reduction allowing for $NT$
```

an individual run. That is, within a run it might be identified that the following set of pairs are not executed in the iteration - $(iPoint_i, iPoint_j)$, $(iPoint_m, iPoint_n)$. For the example, within an individual run $iPoint_i$ and $iPoint_j$ are never executed in the same iteration. These are referred to as an *infeasible iPoint pair*. The same is true for $iPoint_m$ and $iPoint_n$. Two figures are presented. Figure 14 shows many infeasible iPoint pairs had not been eliminated (y-axis) after a given number of runs (x-axis). Only 29 runs are presented as at that was sufficient runs to show there were no infeasible iPoint pairs in the SUT. This does not means the piece of software definitely has no infeasible paths as the testing failed to find a single test vector that resulted in all the loop bounds being maximised.

Figure 15 takes the results for the 29 runs and presents a histogram of how many infeasible iPoint pairs remain for each individual run. The results show a wide range of values but also that the best a single run does in terms of finding infeasible iPoint pairs is 200 remaining. The overall results strongly suggest many runs and iterations are needed to determine that the software does not have any infeasible paths.

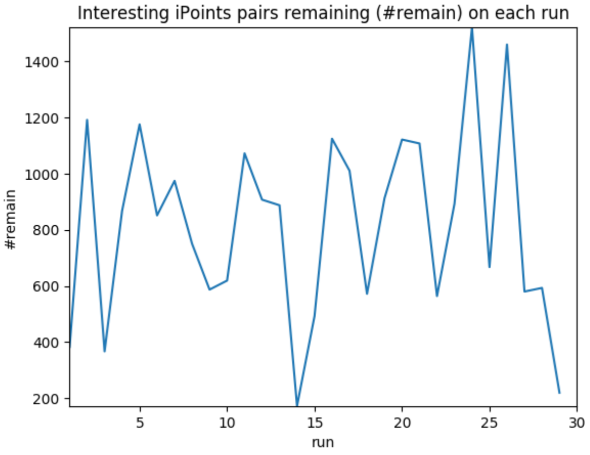

Figure 14: Interesting iPoints Remaining After Each Run

7 CONCLUSIONS

This paper sought to demonstrate a method which identifies when testing is not yielding new information and apply this to a complex platform, the Raspberry Pi 3B+. A variety of methods were examined to achieve this. It was shown that a number of metrics which are commonly assumed to be correlated to the execution time of a program were not guaranteed to be, such as loop iterations. However, other metrics, such as path length, were shown to be correlated with execution time, albeit with some variability due to the complex nature of the platform used.

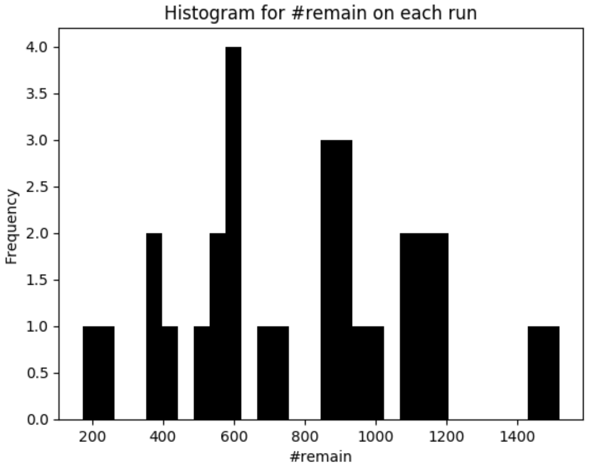

Figure 15: Histogram for Interesting iPoints Remaining After Each Run

The BCHLr testing algorithm was examined in depth using this method, and this paper found that BCHLr does not reliably find the WCET of the SUT on the Raspberry Pi 3B+ platform. (It is noted trials with the other fitness functions used in [19] showed BCHLr was still significantly more reliable than them, however for reasons of space they are not included in this paper.) This is evidenced by the fact that the HWM of BCHLr does not reliably converge over runs using 50,000 iterations, primarily due to complex nature of the software (which is designed with real-time applications in mind) and the hardware platform (which is not a traditional real-time platform). However, this is somewhat expected as BCHLr does not seek to maximise execution time. The fact that it aims to maximise coverage means it should not repeatedly test the longest path. Therefore BCHLr is judged as successful. This result points to using BCHLr to identify candidates for the path yielding the longest execution and then using another algorithm to investigate these further. Future work can explore this idea, and in general how to change the testing approach once further testing is determined to not yield additional information. In Rolls-Royce, RapiTime is used so the soundness of the WCET can be argued in certification [19].

Finally, this paper examined how these results may be used in hybrid-analysis, and methods to highlight potential infeasible paths which otherwise could cause pessimism in hybrid-analysis methods. This approach allows a subset of the program to be presented to an engineer to determine if the potential infeasible paths are truly infeasible, rather than having to examine the entire SUT.

ACKNOWLEDGEMENTS

This work was supported by the HICLASS project, funded by the Aerospace Technology Institute and Innovate UK, as project 113213.

REFERENCES

[1] E. Aarts and J. H. M Korst. *Simulated Annealing and Boltzmann Machines*. John Wiley and Sons, 1989.

[2] A. Agrawal, R. Mancuso, R. Pellizzoni, and G. Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems. In *Real-Time Systems Symposium*, 2018.

[3] M. Bartlett, I. Bate, and D. Kazakov. Guaranteed loop bound identification from program traces for WCET. In *Real-Time and Embedded Technology and Applications Symposium*, pages 287–294, 2009.

[4] M. Bartlett, I. Bate, and D. Kazakov. Accurate determination of loop iterations for worst-case execution time analysis. *IEEE Transactions on Computers*, 59(11):1520 – 1532, nov 2010.

[5] I. Bate and S. Law. Challenges in applying mixed-criticality systems to aircraft engine control systems. In *5th International Workshop on Mixed Criticality Systems*, 2019.

[6] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium*, pages 279–288, 2002.

[7] S. Bünte, M. Zolda, and R. Kirner. Let's get less optimistic in measurement based timing analysis. In *Symposium on Industrial Embedded Systems*, 2011.

[8] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner. Improving the confidence in measurement-based timing analysis. In *Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 144–151, 2011.

[9] A. Burns and S. Edgar. Predicting computation time for advanced processor architectures. In *EUROMICRO Conference on Real-Time Systems*, pages 89–96, 2000.

[10] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[11] S. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean. Open challenges for probabilistic measurement-based worst-case execution time. *Embedded Systems Letters*, 9(3):69 – 72, 2017.

[12] P. Graydon and I. Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, 57(5):759–774, 2014.

[13] D. Griffin, I. Bate, and B. Lesage. Evaluating mixed criticality scheduling algorithms with realistic workloads. In *International Workshop on Mixed Criticality Systems in conjunction with the Real-Time Systems Symposium*, pages 24–29, 2015.

[14] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *Real-Time Systems Symposium*, pages 57–66, 2006.

[15] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement. In *International Conference on Computer Aided Verification*, pages 209–213, 2008.

[16] U. Khan and I. Bate. WCET analysis of modern processors using multi-criteria optimisation. In *Symposium on Search Based Software Engineering*, pages 103–112, 2009.

[17] Usman Khan and Iain Bate. WCET analysis of modern processors using multi-criteria optimisation. *Empirical Software Engineering*, pages 1–24, 2010.

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

[19] S. Law and I. Bate. Achieving appropriate test coverage for reliable measurement-based timing analysis. In *EUROMICRO Conference on Real-Time Systems*, 2016.

[20] T. Lazariv and C. Lehmann. Goodness-of-fit tests for large datasets, 10 2018. https://arxiv.org/pdf/1810.09753.pdf.

[21] B. Lesage, I. Bate, and S. Law. TACO: An industrial case study of test automation for coverage. In *International Conference on Real-Time Networks and Systems*, 2018.

[22] B. Lesage, D. Griffin, I. Bate, and F. Soboczenski. Exploring and understanding multicore interference from observable factors. In *International Conference on Real-Time Networks and Systems*, 2017.

[23] B. Lesage, D. Griffin, I. Bate, F. Soboczenski, and R. Davis. Forecast-based interference: Modelling multicore interference from observable factors. In *Automotive - Safety & Security*, 2017.

[24] S. Lim, Y. Bae, G. Jang, B. Rhee, S. Min, C. Park, H. Shin, K. Park, and C. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[25] F. Massey. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[26] F. Mueller. Generalizing timing predictions to set-associative caches. In *EUROMICRO Workshop of Real-Time Systems*, pages 54–71, Jun 1997.

[27] M. Neil, B. Littlewood, and N. Fenton. Applying bayesian belief networks to systems dependability assessment. In *Safety Critical Systems Club Symposium*, pages 71–94. Springer-Verlag, 1996.

[28] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems Journal*, 5(1):31–62, 1993.

[29] C. Park and A. Shaw. A source level tool for predicting deterministic execution times of programs. Technical Report 89-09-02, Department of Computer Science and Engineering, University of Washington, USA, 1989.

[30] 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*, pages 285–288, 1998.

[31] J. Wegener, H. Sthamer, B. Jones, and D. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

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

[33] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[34] N. Williams. WCET measurement using modified path testing. In *International Workshop On Worst-Case Execution-Time Analysis in conjunction with the EUROMICRO Conference on Real-Time Systems*, 2005.

[35] N. Williams and M. Roger. Test generation strategies to measure worst-case execution time. In *Workshop on Automation of Software Test in conjunction with International Conference on Software Engineering*, pages 88–96. IEEE, 2009.