WCET Analysis of Modern Processors Using Multi-Criteria Optimisation

Usman Khan, Iain Bate
Department of Computer Science, University of York, York, United Kingdom
Email: usmannmkhan@hotmail.com, iain.bate@cs.york.ac.uk

Abstract

The Worst-Case Execution Time (WCET) is an important execution metric for real-time systems, and an accurate estimate for this increases the reliability of subsequent schedulability analysis. Performance enhancing features on modern processors, such as pipelines and caches, however, make it difficult to accurately predict the WCET. One technique for finding the WCET is to use test data generated using search algorithms. Existing work on search-based approaches has been successfully used in both industry and academia based on a single criterion function, the WCET, but only for simple processors. This paper investigates how effective this strategy is for more complex processors and to what extent other criteria help guide the search, e.g. the number of cache misses. Not unexpectedly the work shows no single choice of criteria work best across all problems. Based on the findings recommendations are proposed on which criteria are useful in particular situations.

1 Introduction

The study of real-time systems incorporates all systems which have to respond within a fixed, finite time and where a delayed answer is as bad as a wrong response. More formally, a real-time system is one where "*correctness depends not only on the logical result of the computation, but also on the time at which the results are produced*" [4]. Thus, timeliness is a crucial aspect of a real-time system. One essential measure for all forms of schedulability analysis is the WCET. WCET research has proceeded in two distinct directions.

The method of *static analysis* aims to analyse the hardware and software under test statically i.e. without executing the software, and then use this information to derive an estimate for the WCET. Thus, given a processor architecture and the program to be executed, static analysis works by analysing execution paths and simulating processor characteristics to determine the worst-case path in a program [16, 10]. The worst-case path is the path which produces the maximum run-time. Static analysis has the benefit of guaranteeing a safe upper bound for the WCET making it ideally suited for critical systems but its pessimism and portability can be seen as a weakness for less critical systems [20].

In contrast, the method of *dynamic analysis* takes a measurement-based approach to the task of determining the WCET of a program. Given the program, and the target processor, dynamic analysis executes the program with a large and diverse range of inputs, and measures the execution time of each successive run. This information is used to find the WCET. Often the largest value is taken as the WCET. An exception to this is probabilistic analysis where a statistical estimate is made of the execution time value results to predict a WCET with the required reliability [1].

Dynamic Analysis, however, requires the software under investigation to be tested with a significantly large number of appropriate inputs to achieve a confidence level that the worst-case run has occurred, and therefore, the WCET has been encountered. Two distinctly different approaches exist for dynamic analysis; hybrid approaches which combine static analysis and measurement-based approaches, and so called search-based techniques. The hybrid techniques [1] show significant promise but come with a high degree of complexity [20]. Instead here we explore the extent to which search-based techniques can be improved.

Search-based techniques have been used to generate the input test-data, as the input space of the program can be large and quite complex. Results of these optimisation techniques [13, 15, 16, 17] show that they are effective in sampling large search spaces. They have been applied to both academic benchmarks and more significantly to industrial case studies from both automotive [17] and aerospace applications [15]. In the case of the aerospace application [15], the results were used in preference to those from static analysis which had at least 30% pessimism [5]. Pessimism is defined as the difference between the estimated and actual value. It has been observed that the pessimism produced by static analysis techniques is greater than the optimism linked with evolutionary, search-based techniques [16]. Further, it is widely recognised that the pessimism associated with static analysis will only get worse with more complex processors [20].

In [16] a comparison is performed of these techniques with those of static analysis showing that

978-0-7695-3675-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSBSE.2009.20

each technique has its own merits and drawbacks with the decision of which to use depending very much on the application context. As with all dynamic analysis approaches they can only guarantee a safe upper bound on the WCET if the appropriate test data is used. All the previous work based on search-based techniques have only used a single criteria function, the WCET itself, in the search for good test-inputs. Wegener [13, 16, 17] has suggested adding other criteria into a combined fitness function as a way of improving the search, i.e. the time to find the maximum execution and the magnitude of this execution time. As modern processors have a number of performance-enhancing features such as caches and pipelines, use of these features can significantly alter the time taken by the software. In some cases, these features even cause a different path to emerge as the new worst-case path. Thus, including these features in the fitness function for the search may beneficially modify the search direction. This type of strategy is widely recognised across search-based software engineering [21]. An example of how a feature may be included is to have a criteria for the number of cache misses encountered.

For example, having an additional criterion could be added to influence the search to derive test cases which maximise the number of cache misses. It has the additional benefit that it may increase the number of instructions executed or data accesses made or find most interesting paths through the software. Of course this does not guarantee to find the WCET but may help. To the best of our knowledge no other work has explored the use of multi-criteria approaches. The closest work to this is Betts [2] who proposed the use of different coverage criteria, instead of the more usual Modified Condition/Decision Coverage [12] which is an often used structural coverage criteria. These other criteria allow for low-level processing effects on the WCET, e.g. the order of instructions through a pipeline. However they have not evaluated the concept, and using coverage criteria is a different approach to what is proposed here, as we do not attempt to achieve complete coverage.

The contribution of this paper, consequently, is to establish the processor and software features which have the most notable impact on the search for the WCET, and to determine the best way in which the effect of these features can be captured in a multi-criteria heuristic function.

The structure of the paper is as follows. Section 2 surveys issues identified in the literature that affect WCET analysis. Next, different methods for WCET analysis using search-based techniques are proposed. In section 4 the results are presented. Finally conclusions are drawn in section 5.

2 Problems in Timing Analysis

A number of problems are encountered in analysing the temporal behaviour of real-time programs. These range from complexity in the real-time programs to be analysed, to the internal features of the hardware. These are discussed in the following sections.

2.1 Causes of Complexity in Real-Time Programs

Groβ [8] defines some aspects of complexity in real-time programs which make it difficult for a dynamic analysis technique, for example a search-based testing method, to *precisely* predict the WCET. The following are identified as important criteria which add to a program's complexity, and consequently, cause difficulty for search-based testing to find the actual solution:

1. High nesting within a program
2. Low path probability of paths within a program, where the low probability is caused by very few values in the domain of the input variable(s), for the program being analysed, leading to the choice of that path.
3. High parameter and algorithmic interdependence, which is caused by the execution time being highly dependent on the values of the input variables, e.g. in a parameter-dependent loop.
4. Size of the program's input, or the range values that input variables can take.

Groβ [8] validates the difficulty of each of these measures by applying them individually to 22 simple test programs. The results show that, generally, increasing the difficulty of each measure makes it more difficult to find the actual WCET. Even though the complexity measures have only been applied to simple test programs, Groβ [8] proposes that the significant number of test programs used, together with the considerable variations in the complexity of the test objects, makes the results generally applicable to larger more complex real-time programs.

2.2 Complexity in Processor Hardware

As with program characteristics there are certain hardware features that make finding the WCET analysis more difficult. These are widely recognised to be as follows [20].

1. Cache – there are three types of cache: instruction, data and unified as well as many different configurations, e.g. direct mapped, set associative or multi-level. Normally as the number of cache misses increases so does the WCET.
2. Pipeline – here three types of complexity are introduced; instruction level parallelism, resource sharing and allocation, and dynamic scheduling where instructions can be executed out of order.
3. Branch prediction – similar to caches there is an aspect of global history to decide which instructions' information is stored for. However in addition there can be complex logic deciding the result of the prediction.

However there are other significant issues that these features introduce. The principal one being timing anomalies which can be introduced when processors feature dynamic scheduling [11]. Timing anomalies are defined as situations where the counter-intuitive influence of the local execution time of one instruction has an adverse bearing on the global execution time of the whole task [20]. Thus, a faster execution within part of the code can actually cause an increase in the execution time of the whole task, perhaps even leading to the WCET.

An advantage of dynamic analysis techniques, including the search-based techniques discussed here, is the fact that timing anomalies are allowed for in the measures of the actual run-time of the software on the target processor [1]. Thus, an execution containing a timing anomaly but with a larger overall execution time is still considered, and given preference over a non-anomalous execution.

3 Finding the WCET Using Search

In section 2 a number of contributing factors were identified for why WCET analysis is a difficult, generally speaking intractable [20], problem. Each of these factors may also influence the final WCET, e.g. maximising the number of cache misses normally results in a larger execution time. They also mean that there is no way of knowing what the actual WCET is. Therefore the normal approach [20] is to consider how different analysis methods affect the ability to find the WCET. As with all approaches to WCET analysis, it is assumed the software being analysed runs non-preemptively with effects from other software (e.g. context switches, cache pollution etc) being accounted for as part of higher-level schedulability analysis [20].

Two alternative approaches were considered. Firstly to perform a comprehensive set of evaluations and determine which approach best achieves our desirable properties, i.e. achieving the maximum execution time, repeatability / reliability, and efficiency (i.e. how long the search takes), and in what circumstances. Secondly, to again perform comprehensive evaluations but use an approach such as Principal Component Analysis (PCA) [9] to try and determine dominant patterns. The first approach was taken as it was anticipated that the combination of a large complex problem landscape and the fact the landscape may be different for each problem will make it difficult for PCA to be applied. However with the results from the first approach in place, future work may then apply PCA as it could be used in a more focussed way.

There are four phases to the work:

1. Single criterion – in essence perform WCET analysis as previously demonstrated by Tracey [15], Wegener [16,17] and Groβ [8].
2. Low-level analysis – investigate to what extent adding criteria based on low-level (instruction) features influences the search. Key factors are the number of cache misses and branch mis-predictions. Cache misses can be separated into data and instruction or treated together.
3. High-level analysis – examine to what extent high-level (program flow) features affect the search. The key factor here is the loop count.
4. Integrated analysis – consider how combinations of the previous three phases perform.

The following sub-sections discuss the phases in more detail.

3.1 Phases 1 – Single criterion search

A Genetic Algorithm (GA) was developed for automatically generating input test-data. The algorithm was chosen as previous work [15] has used it, albeit it with minor differences in approaches, and initial trials have shown it to be effective. This algorithm initially uses execution time as its single fitness measure. The representation, that is subsequently manipulated through crossover and mutation, is the set of values for the input variables. This data is the only item under our control.

The design of the GA used is a relatively general one based on those in [18]. Certain common choices and strategies for the algorithm were, however, fixed and used throughout this project to support fair comparison between the different approaches. These choices are:

1. Population Size: 100
2. Number of Generations: 100
3. No. of elitist children in the next generation: 1
4. Selection: Roulette Selection
5. Crossover Strategy: Arithmetic Crossover
6. Mutation Strategy: Random Mutation
7. Crossover Percentage: 60%
8. No. of runs for each experiment: 10

A choice of 100 for the population size ensures that there is sufficient diversity within the population without this value being so large as to significantly slow the computations at each generation. Further, fixing the number of generations to 100 ensures that the trajectory of the search can be examined once it has completed, without being so large as to significantly slow down each experiment. Restricting the number of elitist children in the next generation to 1 allows only the best solution to be carried over from one generation to the next, rather than a large number of best solutions. This helps maintain the diversity of the search. Additionally, roulette selection, arithmetic crossover and random mutation are common strategies adopted by the genetic algorithm, which work well in practice and are simple to implement. A crossover percentage of 60% has also been observed to work well in a genetic algorithm. The last requirement, for each experiment to be conducted 10 times, enables the reliability of the solution, i.e. the degree of repeatability, to be recorded. Here reliability is defined as the likelihood of obtaining similar results in practice.

Simplescalar [3] was chosen as the processor simulator on which the working of the input program would be analysed. Further, the ARM-processor was chosen for the experiments as it was supported by Simplescalar and as configured in this work represents a relatively complex processor, containing two-levels of cache, branch prediction and out-of-order execution. This ensured that, in general, experimental results were representative of executions on a modern processor. Specific details of the processor used are as follows. These are the default configuration of simplescalar and as such has often been used in other work [14].

- Branch predictor: bimodal
- Branch predictor table size: 2048 entries
- Branch Target Buffer (BTB): 512 blocks
- Data Cache (L1): 128 blocks, 32 bytes block size, Least Recently Used (LRU) replacement policy
- Data Cache (L1) Hit Latency: 1 cycle
- Instruction Cache (L1): 512 blocks, 32 bytes block size, LRU replacement policy
- Instruction Cache (L1) Hit Latency: 1 cycle
- Memory Access Latency: 18 cycles (first block in a multi-fetch instruction), 2 cycles (subsequent blocks)

3.2 Phase 2 – Low-level analysis

Based on a detailed consideration of the low-level microprocessor features that affect the WCET, summarised in section 2, the following is a list of criteria considered as part of a multi-criteria search method. Each of the criteria can be used in isolation or combined with others. For example it may be expected to use combined knowledge of cache misses, i.e. an overall total for the data and instruction cache misses. Number of misses is used instead of a normalised rate (total misses divided by the number of memory accesses) as this will favour software paths with more memory accesses. These figures are obtained by parsing the detailed log files, produced by Simplescalar, for the information needed. Future work could consider other means for obtaining the information, including *in vivo* analysis.

1. Execution Time (ET)
2. Branch Prediction Misses (BPM)
3. Data (Level 1) Cache Misses (DCM)
4. Instruction (Level 1) Cache Misses (ICM)

These measures were used to create the following heuristics. The heuristics feature different ratios (or weightings) used to combine the results from the evaluation of each of the criterion. Different ratios are used between the same sets of criteria so that the effect of biasing can be examined. The ratios were chosen based on the results of some preliminary assessments to determine the typical quantities involved and then to provide some degree of balancing. For example the ratios between BPM and ICM is chosen so the outputs of their respective analyses tends to give a similar magnitude.

(a) BPM only
(b) DCM only
(c) ICM only
(d) DCM and ICM in the ratio 1:1
(e) DCM and ICM in the ratio 3:1
(f) DCM and Instruction Cache Accesses in the ratio 1:1
(g) BPM, ICM and DCM in the ratio 2:1:1
(h) ET and BPM in the ratio 1:10
(i) ET and DCM in the ratio 1:10
(j) ET and ICM in the ratio 1:10
(k) ET, DCM and ICM in the ratio 1:5:5
(l) ET, DCM and ICM in the ratio 2:15:5
(m) ET, BPM, DCM and ICM in the ratio 1:10:10:10
(n) ET, BPM, DCM and ICM in the ratio 7:10:10:10
(o) ET, BPM, DCM and ICM in the ratio 7:1:1:1

An important choice when gathering the metrics for each of the criteria is whether they are measured for the whole program or at specific points. For example, the cache miss rate could be measured separately for each memory access and each measure represented by its own criterion. However this would result in a large number of criteria even for small programs and it is generally accepted that searching across more than six criteria is difficult [6]. For this reason one overall measure (a combination by a weighted sum, using the previously mentioned ratios as weights, of the results from evaluating each individual criteria) is made for the whole program.

3.3 Phase 3 – High-level analysis

As previously stated in section 2, the principal issue affecting the WCET is the number of loop iterations and hence this is chosen as a criterion. In a similar fashion to the criteria for low-level analysis, the metrics are gathered across the whole program.

3.4 Phase 4 – Integrated Analysis

The heuristics used as the fitness functions for this phase were obtained by combining the heuristics from Phases 2 and 3. However, as Phase 3 only used a single new execution measure, the number of loop iterations, the resulting heuristics merely added the number of loop iterations to each of the heuristics developed in Phase 2. The heuristics used for the Integration Phase, thus, consisted of:

(a) BPM and Loop Iterations (LI) in the ratio 10:1
(b) DCM and LI in the ratio 10:1
(c) ICM and LI in the ratio 10:1
(d) DCM, ICM and LI in the ratio 5:5:1
(e) DCM, ICM and LI in the ratio 15:5:2
(f) DCM, ICM and LI in the ratio 5:5:1
(g) BPM, DCM, ICM and LI in the ratio 10:5:5:2
(h) ET, BPM and LI in the ratio 1:10:2
(i) ET, DCM and LI in the ratio 1:10:2
(j) ET, ICM and LI in the ratio 1:10:2
(k) ET, DCM, BPM and LI in the ratio 1:5:5:2
(l) ET, DCM, ICM and LI in the ratio 2:15:5:4
(m) ET, DCM, BPM, ICM and LI in the ratio 1:10:10:10:4
(n) ET, DCM, BPM, ICM and LI in the ratio 7:10:10:10:10

4 Evaluation

A set of benchmark problems is publicly maintained by the Mälardalen WCET research group [7]. This comprises a number of programs used to evaluate the performance of different WCET analysis tools. As the benchmarks have already been categorised by type then a selection of programs were chosen to give a reasonable sample from each category after discounting the more trivial examples, e.g. small pieces of code with no distinctive features. Consequently sixteen of the programs, listed in Table 1, were chosen that had a range of characteristics.

Benchmark Programs	WCET (mean)	WCET (max)
Factorial	2,053.3	2,188.0
Cover	3,991.0	3,991.0
Insertion Sort (10 inputs)	1,328.8	1,333.0
Insertion Sort (50 inputs)	17,528.2	18,240.0
Insertion Sort (100 inputs)	59,367.9	63,216.0
Discrete Cosine Transformation (DCT)	4,186.0	4,186.0
Extended Petri Net Simulation (PETRI)	16,722.7	18,378.0
Matrix multiplication	21,376.0	21,376.0
Quadratic Equations Root Computation (QERC)	1,069.0	1,069.0
Janne Complex	437.3	443.0
Matrix Inversion	3,787.0	3,787.0
Computing an exponential integral function (EXP)	14,317.5	16,358.0
Quick Sort	2,905.4	2,980.0
Fast DCT (FDCT)	4,712.0	4,712.0
Fast Fourier Transformation (FFT)	14,026.2	14,449.0
Select	10,711.7	10,757.0
Statistics Program	14,429.7	14,470.0
Binary Search	269.0	269.0

Table 1 - Phase 1 Results (units = clock cycles)

Each experiment was run 10 times, in order to evaluate the reliability of the solution produced. Reliability is assessed based on the variance of the resulting execution times. Further, the WCET estimate predicted at the end of 100 generations, with a population size of 100, was recorded, together with the time taken to produce the estimate, in order to evaluate the direction, quality and efficiency of the search.

4.1 Phase 1

The results of the experiments are presented in Table 1 for the WCET estimate produced for each benchmark program. These results show the quality of the solution generated for each program, using execution time as the fitness value, and is, additionally, a measure of the general direction of the search after 100 generations. A higher estimate of the WCET is considered more accurate and therefore, better.

The Phase 1 results for the reliability of the search show that three programs, *Insertion Sort, with 100 inputs, PETRI* and EXP had large variances, (denoted by the square of the difference between the mean WCET and the WCET from each of the 10 repeated trials), and so, the results of these programs were the least reliable. At the other end of the scale, seven programs, e.g. *Cover*, had zero variance and are classed as being simple to examine. The variances, and thus the reliability, of other programs varies from under 100 cycles^2 for *Insertion Sort, with 10 inputs* and *Janne Complex* to almost a million cycles^2 for *Insertion Sort, with 50 inputs*. It should be noted that whilst a million cycles^2 may seem large that this only corresponds to a standard deviation of a thousand cycles which is just over 5% of the WCET(max).

4.2 Phases 2

As a summary for the whole set of benchmarks it was found in eleven of the eighteen cases a single criterion, the execution time, as used in phase 1 gave the best results. However this leaves seven important cases for which it did not perform as well. For reasons of space, a full set of results are not presented in depth only those with more interesting characteristics. Interesting is defined as those for which different criteria made a significant difference in terms of quality, efficiency or reliability. The more interesting benchmarks are insertion sort, factorial, janne complex and quadratic.

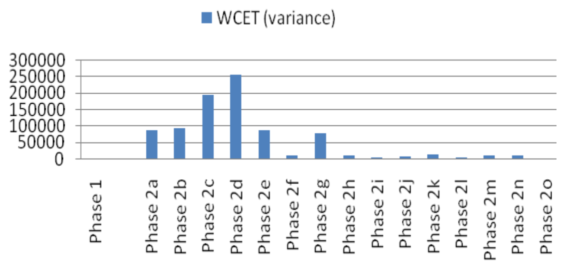

Figure 1- Insertion Sort (10 inputs) (Reliability of Solution)

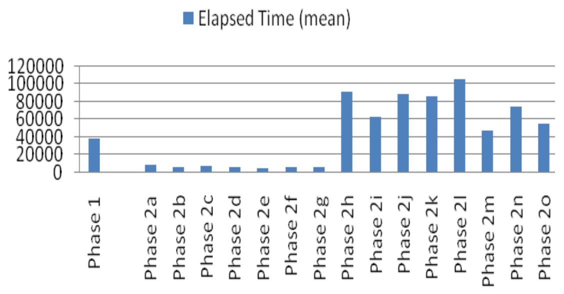

Figure 2 - Insertion Sort (100 inputs) (Efficiency of Solution)

The results of the experiments, for the *interesting benchmarks*, are presented in Figures 1-8, where Phases 2a - 2o represent the heuristics (a) – (o) in section 3.2. The results show that, in general single low-level fitness criteria, for example, only branch mispredictions, data cache misses or instruction cache misses, do not perform well in practice. Thus, in Figures 3-8, the WCET estimate produced by Phases 2a, 2b and 2c ('WCET (max)' for each of these phases), are all among the lowest values predicted. Figure 1 (units for the y-axis are clock cycles^2) shows that these results also have a large variance, and consequently, low reliability as well, thereby reducing their usefulness. The speed of producing these results, however, is high as represented in Figure 2 (units for the y-axis are seconds of actual compute time on a 2.4 GHz Intel processor). Thus, a quick but exceedingly inaccurate estimate for the WCET can be obtained using these low-level criteria individually as the fitness measure. This trend was further demonstrated across the whole set of benchmarks. However if speed and quality are concerns, then the single criteria of execution time normally offers the best choice.

However, as previously stated, in 7 of the benchmark programs, a multi-criteria fitness measure produced a higher-quality solution than execution time alone. This shows that multi-criteria heuristic fitness functions can still be gainfully used in practice for a program, if appropriate ones are chosen. For example, the Factorial program calls itself recursively passing data at each recursive call. This implies that the program makes heavy use of the data cache, and therefore, the number of data cache misses, and latency caused as a result of these misses, can be a useful metric in guiding the search to the WCET. The results of the Phase 2 experiments on the Factorial program are presented in Figure 5. These results confirm the analysis about the dependence of the Factorial program on the data cache, as data cache misses used alone as the fitness measure (Phase 2b) produces a solution whose quality is only slightly poorer than that of the solution produced by execution time as the single fitness measure (Phase 1). Further, a combination of execution time and data cache misses (Phase 2i) as the fitness measure produces a better solution than execution time used alone, and the highest-quality solution overall, confirming the analysis. It is noted though the wrong combination of criteria has a negative effect as shown by the results of Phase 2m for instance.

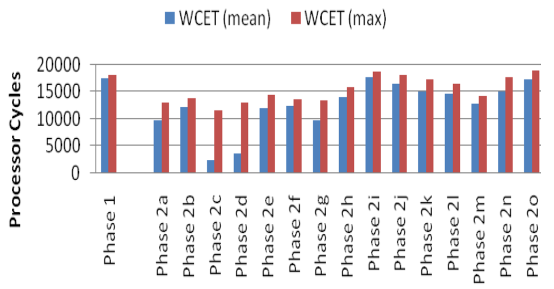

Figure 3 - Insertion Sort (50 inputs) (Quality of Solution)

A program's dependence on the data cache increases with the size of its inputs and the number of calculations performed within it. This information can be gainfully used in devising the best heuristic fitness function to aid the search in reaching the WCET. Thus, a combination of execution time and data cache misses works well in practice, for a program taking a large number of inputs. For example, the Insertion Sort program with 10 inputs (Figure 6), has a relatively low dependence on the data cache. Thus, execution time as the single fitness measure produces a higher quality solution than execution time used with the total number of data cache misses. However, as the size of the inputs is increased, the dependence on the data cache becomes prominent. For example, in the case of the Insertion Sort program with 50 inputs (Figure 3), the combination of data cache misses and execution time as the fitness measure, gives a higher-quality solution than execution time alone, and only a marginally poorer solution than execution time, branch prediction misses, data cache misses and instruction cache misses used altogether as the fitness function, which produces the overall highest-quality solution. However, in the Insertion Sort program with 100 inputs (Figure 7), this situation is reversed. In this program, execution time and data cache misses used together as the fitness function finds the highest-quality solution, thus, validating the analysis. Moreover, the results are produced in only a slightly less efficient way than execution time, branch prediction misses, data cache misses and instruction cache misses used together as the fitness function (Phase 2o), confirming the applicability of this joint fitness measure.

The presence of a large number of loops or conditional statements within a program can cause instruction cache misses, as the instructions after the target of a conditional branch will need to be loaded into the instruction cache, and, depending on the branch prediction method used, cause branch mispredictions as well. Thus, these two execution measures can be gainfully used for such programs as additional criteria within the fitness function of the genetic algorithm. For example, the Quadratic Equations Root Computation program takes only 3 inputs. However, these are used in a loop and conditional statements within the program. Thus, instruction cache misses and branch mispredictions can be used to aid the search. The results for this program (Figure 8) show that execution time and branch mispredictions together (Phase 2h), and execution time and instruction cache misses together as fitness functions (Phase 2j) produce higher-quality solutions than execution time alone, with execution time and instruction cache misses together producing the highest-quality solution. Further, a combination of execution time, branch mispredictions, instruction cache misses and data cache misses as the fitness measure also attains a higher-quality solution than execution time alone. These results show that branch mispredictions and in particular, instruction cache misses can be gainfully used as additional fitness criteria in programs with similar characteristics.

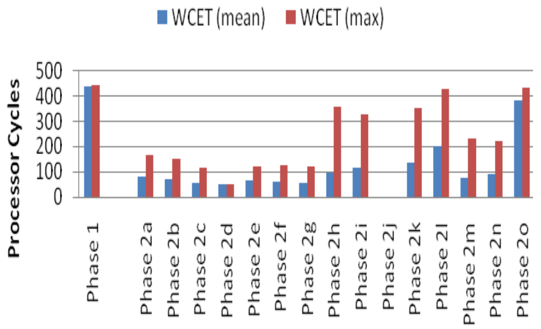

Figure 4 - Janne Complex (Quality of Solution)

In some cases, however, the addition of extra criteria as fitnesses confuses the search algorithm. This is particularly true when the criteria assign conflicting fitness values to a program execution. For example, the program Janne Complex contains a two-level nested loop. Entry into this loop, is however, dependent on the input values and only a small number of input values cause this loop to be executed. Thus, the path probability of this loop is low. Consequently, if the input values do not permit entry into the loop, a significant number of cache misses outside the loop can confuse the search into giving a high fitness to these program inputs. Thus, executing the loop will then be regarded as an undesirable proposition for the search, as a high fitness solution did not execute it. Clearly with careful tuning of weightings a different result could be achieved, however this is out of scope of this paper. As a result, the loop will remain unexecuted and the WCET estimate produced may be inaccurate. Figure 4 shows the results of applying the search to the Janne Complex program. The result shows that execution time as the single fitness measure (Phase 1) produces the highest-quality solution, while the result for a combination of execution time, branch mispredictions, data cache misses and instruction cache misses, with a heavy bias towards the execution time, as the fitness measure (Phase 2o) is only marginally poorer. However, the rest of the results range from only slightly poorer to much worse. This shows that the fitness assigned to other criteria, such as cache misses and branch mispredictions may cause program inputs resulting in only a few iterations of the loop to be assigned a higher fitness than inputs which execute the loop longer, thereby reducing the program inputs that execute the loop longer from the genetic algorithm's population. The results for Phase 2j show that the highest fitness is dominated by instruction cache misses to the extent that a zero loop count had the highest fitness even though the execution time was low.

Thus, the conflict between criteria is a serious problem in multi-criteria fitness functions as it can inhibit, rather than aid, the search, thus, preventing it from finding the best solution. This problem can be resolved by using the proposed method of program analysis before the search, in order to evaluate the criterion, or combination of criteria that best guides a search to the program's WCET. Later in the paper this is explored further.

4.3 Phases 3 and 4

The results of the experiments are shown in Figures 5-8. The results show that, in general, combining the criteria from phases 2 and 3 do not work well in practice. However this does not preclude the possibility that for other programs the trend will be reversed. The results from Phase 4, additionally, take much longer as they extract a larger amount of execution information, and thus, have a fairly poor speed and efficiency. The reliability of these results, although varying significantly from one program to another, is generally fair and comparable to the results from the other phases.

Moreover, of the remaining 8 programs where execution time does not find the highest-quality solution, in 3 programs, the solution found by execution time is within 1% of the highest-quality solution while, in a further 2 programs, the solution found by execution time is within 5% of the highest-quality solution. The maximum number of highest-quality solutions found by a fitness measure, other than execution time, is 8 out of 18. This is achieved by using execution time, branch prediction misses, data cache misses and instruction cache misses together, with a heavy weight assigned to execution time, as the fitness measure (Phase 2o). Similarly, a combination of execution time and loop counts as the fitness measure (Phase 3b) finds the highest quality solution in 8 (different) cases out of the 18 highest-quality solutions. This shows the general suitability, and applicability, of these fitness measures. However, program analysis should be done beforehand to determine the programs on which use of this particular heuristic as the fitness measure is likely to guide the search to the WCET.

For example, loop iterations (Phase 3a) are seen as the best fitness measure for the Factorial program in Figure 5, as they lead the search to the highest-quality solution. Thus, program analysis beforehand can establish that the number of procedure calls is directly proportional to the execution time and directly dependent on the program inputs. Consequently, use of the number of calls, which subsumes the number of loop iterations (as recursive calls can be considered as loops for the purpose of program analysis) as the fitness heuristic, is the measure most likely to lead a search to the program's WCET.

For the Insertion Sort program, thus, a prior program analysis can establish that loop counts have the most significant bearing for this program, as the program simply consists of a nested loop. Further, the number of iterations of the loop is not dependent on the number of inputs but, rather, the ordering of the inputs. Thus, while increasing the number of inputs will increase the number of data cache accesses and, consequently, data cache misses, the number of times the loop iterates has a more significant bearing on the execution time than the number of data cache misses. Additionally, problems have been seen in the guidance given to the search when two or more criteria are combined. Consequently, it is recommended that only the number of loop iterations, either on its own (Phase 3a) or in combination with execution time (Phase 3b), is used to guide the WCET search. The results show that this analysis holds, as the fittest solution produced by either of these 2 criteria produces the highest-quality solution or a solution whose execution time is within 5% of the best overall estimate for the WCET amongst all the experiments. This is demonstrated in Figure 4, which show high-quality solutions, and other analysis demonstrated it to have an excellent overall reliability.

In contrast, in a program such as Quadratic Equations Root Computation, which contains an input-data dependent loop and input-data dependent conditional statements, the execution time of the loop is not the primary contributor to the overall execution time. Instead the presence of a loop and conditional statements jointly imply that branch mispredictions or instruction cache misses are likely to affect a significant effect on the execution times. Thus, either of these two measures, when combined with execution time in order to ensure that the search does not proceed in a wrong direction, can be used as an effective fitness measure. The results (Figure 8) show that this holds in practice.

4.4 Proposed Heuristics

Using the results from Phases 1, 2, 3 and 4, the following fitness heuristic is proposed (where the first matching value should be used as the heuristic):

1. If the program has a single path through it, i.e. no conditional statements or loops are dependent on the program inputs, then execution time should be used as the single fitness measure.
2. If the program contains a large number of input-data dependent loops, particularly deeply-nested loops, or an input-data dependent number of self-recursive procedure calls, and few or no conditional statements, then the number of loop iterations and measured execution time should be jointly used as the fitness function. (Large in this context is measured by the percentage of the source code contained within a loop, with a proposed value of 75% or greater constituting a large number of loops within a program.)
3. If the size of the program's input space is large, i.e. the program takes a large number of inputs, and there are multiple paths within the program, where the choice of path is dependent on the program inputs, then data cache misses and execution time should be used together as the fitness function. (In this context, a large number of inputs is defined as the ratio of the size of the inputs to the size of the data cache. If this ratio exceeds a proposed measure of 25%, then the fitness measure proposed in this step should be used.)
4. If the program contains a large number of conditional statements or loops, where the condition is dependent on the program inputs, then instruction cache misses and execution time together should constitute the fitness function. (A large number of conditions is defined as the ratio of the number of conditions to the total number of lines in the source code. A value of 15% or over would constitute large in this context.)
5. If the program contains large basic blocks, that take a long time to execute, then execution time should be used as the fitness measure.
6. If neither of the preceding steps matches the program's characteristics, then execution time should be utilised as the fitness measure.

The choice of heuristic assumes that it will be possible to do program analysis to find the significant characteristics of the program before using a search algorithm on the program to determine its WCET. Such an analysis would benefit the subsequent search significantly. However, if it is not possible to do this analysis, this research recommends the use of the measured program execution time as the fitness measure, as it has been found to be the best-performing general-purpose fitness measure for generating a high-quality (result compared to the best found), reliable (variance of final result over 10 trials) and efficient (time taken to search compared to search with a single objective of execution time) estimate for the WCET.

Benchmark Program	**FFP**	**Performance (%)**		
		Quality	**Reliability**	**Efficiency**
Factorial	L,E	97.6	164.7	518.5
Cover	E	100.0	100.0	100.0
Insertion Sort (10 inputs)	L,E	100.0	323.5	173.1
Insertion Sort (50 inputs)	L,E	97.9	411.5	81.9
Insertion Sort (100 inputs)	L,E	97.6	200.0	58.2
DCT	E	100.0	100.0	100.0
PETRI	L,E	100.0	100.0	17.6
Matrix multiplication	E	100.0	100.0	100.0
QERC	I,E	100.0	100.0	100.0
Janne Complex	L,E	100.0	331.3	20.6
Matrix Inversion	D,E	99.6	144041.7	93.6
EXP	E	80.4	5,621.3	187.7
Quick Sort	L,E	99.6	195.8	30.0
FDCT	E	100.0	100.0	100.0
FFT	E	100.0	100.0	100.0
Select	L,E	99.9	217.6	212.3
Statistics Program	D,E	99.8	1,088.6	97.5
Binary Search	L,E	100.0	100.0	91.5
Overall		**98.5**	**8,522.0**	**121.3**

Table 2 - Testing the Proposed Heuristics

The proposed fitness function for each benchmark program is listed in Table 2 which also gives the results for the performance metrics (quality / accuracy, reliability and efficiency) and the Fitness Function Proposed (FFP), where L denotes loop count, I the instruction cache misses, D the data cache misses and E the execution time. The quality, reliability and efficiency of the proposed fitness in this table are measured in comparison to the highest-quality solution produced for the respective benchmark problems. In contrast to earlier in the paper, in Table 2 reliability is computed by variance of the highest-quality solution divided by the variance found with the proposed fitness function. Efficiency is computed by time taken to find the highest quality solution divided by the time taken with the proposed fitness function.

5 Conclusions

The paper shows how existing work on search-based WCET analysis can be extended. However the choice of criteria is not always straightforward. In particular the work has showed that simply introducing a wide range of criteria gives bad results and no single set of criteria works across all the problems. Based on the detailed evaluation performed, recommendations are formed and shown to be effective via further evaluation.

References

[1] G. Bernat, A. Colin and S. Petters, WCET Analysis of Probabilistic Hard Real-Time Systems. In: Proceedings of 23rd IEEE Real-Time Systems Symposium, pp. 279-288, 2002.

[2] A. Betts, G. Bernat, Raimund Kirner, Peter Puschner, and Ingomar Wenzel, WCET Coverage for Pipelines, Techincal report for the ARTIST2 Network of Excellence, August 2006.

[3] D. Burger and T. Austin, The Simplescalar tool set, version 2.0. SIGARCH Computer Architecture News. 25(3), 13-25, 1997.

[4] A. Burns, and A. Wellings, Real-Time Systems and Programming Languages, 3rd Edition, Addison Wesley 2001.

[5] R. Chapman, Static Timing Analysis and Program Proof, PhD thesis, University of York, 1995.

[6] C. Coello, A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. Knowledge and Information Systems. 1(3), 269-308, 1999.

[7] A. Ermedahl and J. Gustafsson, WCET Project / Benchmarks. Accessed: 5 May 2008. Available at: www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[8] H. Groβ, Measuring Evolutionary Testability of Real-Time Software. Ph.D. thesis, University of Glamorgan/Prifysgol, 2000.

[9] I. Jolliffe, Principal Component Analysis, Wiley, 2005.

[10] R. Kirner, P. Puschner and I. Wenzel, Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. In Proceedings of the 4th Euromicro Workshop on Worst Case Execution Time Analysis, 2004.

[11] T. Lunqvist and P. Stenstrom, Timing Anomalies in Dynamically Scheduled Microprocessors, In: Proceedings of the 20th IEEE Real-Time Systems Symposium, pp. 12-21, 1999.

[12] P. McMinn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability, 14(2), pp. 105-156, 2004.

[13] H. Pohlheim and J. Wegener, Testing the Temporal Behavior of Real-Time Software Modules using Extended Evolutionary Algorithms. In Proceedings of Genetic and Evolutionary Computation Conference, 1999.

[14] L. Tan, The Worst Case Execution Time Tool Challenge 2006: The External Test, 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006), pp. 241-248, 2006

[15] N. Tracey, J. Clark, and K. Mander, The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach, In Proceedings of The International Workshop on Dependable Computing and Its Applications, 1998.

[16] J. Wegener and F. Mueller, A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. Real-Time Systems Journal, 21(3), 241–268, 2001.

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

[18] D. Whitley, A Genetic Algorithm Tutorial. Statistics and Computing, 4, 65-85, 1994.

[19] D. Whitley, Genetic Algorithms and Evolutionary Computing. Van Nostrand, 2002.

[20] 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, P. and P. Stenstrom, The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools, ACM Transactions on Embedded Computing Systems, 7(3), 1-53, 2008.

[21] M. Harman, The current state and future of Search Based Software Engineering, In Proceedings of the Future of Software Engineering 2007, pp. 342-357, 2007.

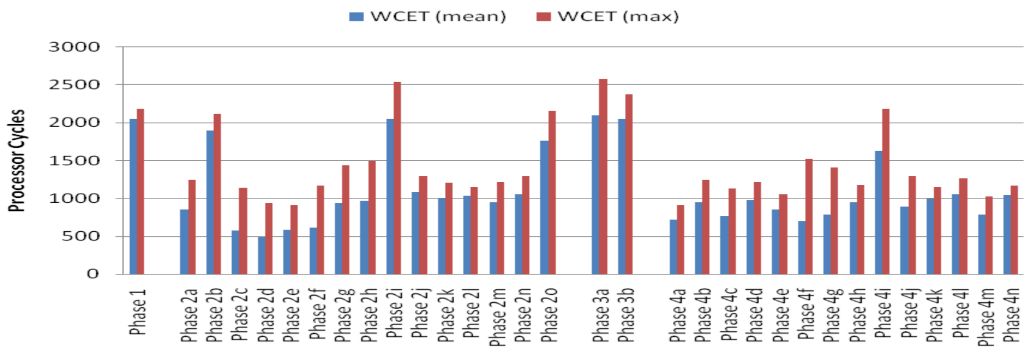

Figure 5 - Factorial (Quality of Solution)

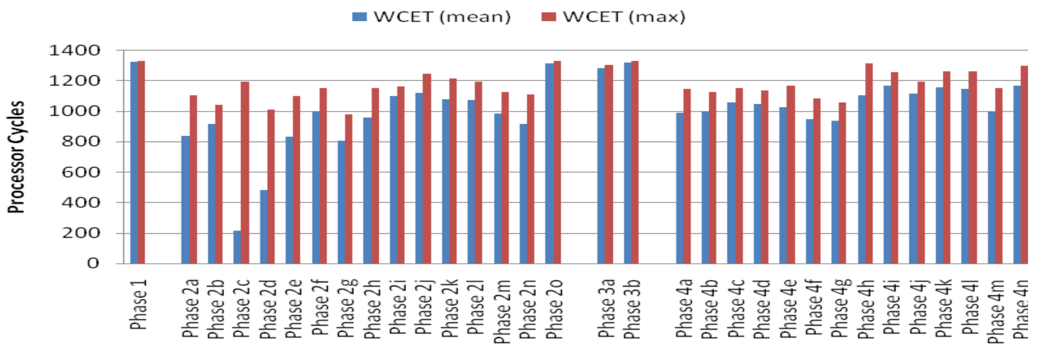

Figure 6 - Insertion Sort (10 inputs) (Quality of Solution)

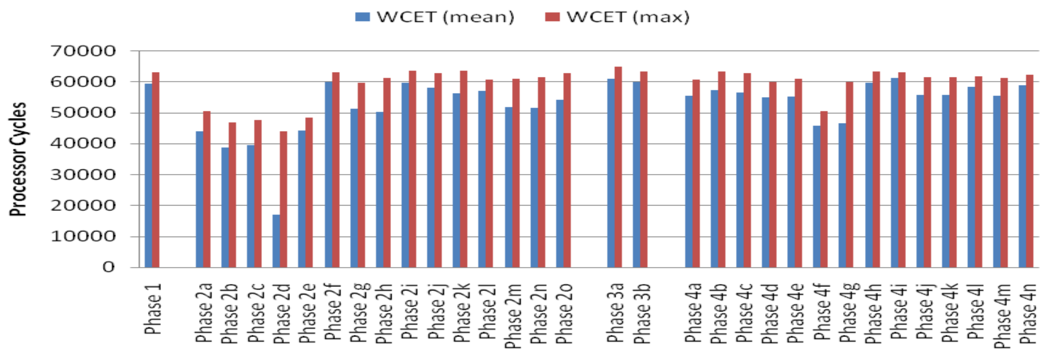

Figure 7 - Insertion Sort (100 inputs) (Quality of Solution)

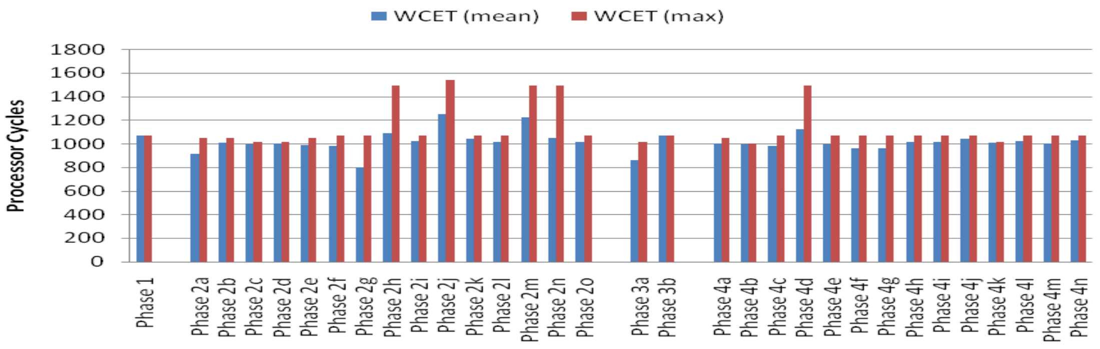

Figure 8 - Quadratic Equations Root Computation (Quality of Solution)