Empir Software Eng (2011) 16:5–28
DOI 10.1007/s10664-010-9133-9

WCET analysis of modern processors using multi-criteria optimisation

Iain Bate · Usman Khan

Published online: 8 June 2010
© Springer Science+Business Media, LLC 2010
Editors: Massimiliano Di Penta and Simon Poulding

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.

Keywords WCET analysis · Multi-criteria optimisation · Search-based software testing

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*" (Burns and Wellings 2001). 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

I. Bate (✉)
Department of Computer Science, University of York, York, UK
e-mail: iain.bate@cs.york.ac.uk

U. Khan
Computer Laboratory, University of Cambridge, Cambridge, UK
e-mail: usman.khan@cl.cam.ac.uk

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 (Kirner et al. 2004; Wegener and Mueller 2001). 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 (Wilhelm et al. 2008).

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 (Bernat et al. 2002).

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 (Bernat et al. 2002) show significant promise but come with a high degree of complexity (Wilhelm et al. 2008). 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 (Pohlheim and Wegener 1999; Tracey et al. 1998; Wegener and Mueller 2001; Wegener et al. 1997) 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 (Wegener et al. 1997) and aerospace applications (Tracey et al. 1998). In the case of the aerospace application (Tracey et al. 1998), the results were used in preference to those from static analysis which had at least 30% pessimism (Chapman 1995). 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 (Wegener and Mueller 2001). Further, it is widely recognised that the pessimism associated with static analysis will only get worse with more complex processors (Wilhelm et al. 2008).

In (Wegener and Mueller 2001) a comparison is performed of these techniques with those of static analysis showing that 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. Pohlheim and Wegener (1999, Wegener and Mueller 2001, Wegener et al. 1997) 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 (Harman 2007). An example of how a feature may be included is to have a criteria for the number of cache misses encountered.

For example, an additional criterion could be added to influence the search to derive test cases which maximise the number of cache misses. This has the added 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 provide a guarantee of finding 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 et al. (2006) who proposed the use of different coverage criteria, instead of the more usual Modified Condition/Decision Coverage (MCDC) (McMinn 2004) which is an often used structural coverage criteria. These other criteria allow low-level processing effects on the WCET to be captured, e.g. the order of instructions through a pipeline. However they have not evaluated the concept. Using coverage criteria is a different approach to what is proposed here, as we do not attempt to achieve complete coverage, instead effort is concentrated on maximising the primary objective—WCET. Other work (Lammermann et al. 2008) has shown that structure-oriented coverage metrics are often not a good indicator as to the quality of the testing output. In addition significant issues are envisaged with achieving this level of coverage as MCDC itself is hard to achieve and the coverage criteria proposed by Betts et al. (2006) would require many more test cases.

The specific goals of this work are to understand how criteria, other than WCET, helps guide the search towards a better overall result and how the results depend on specific applications. Here a better overall result is not only the final value obtained but also the reliability (i.e. the variance in the results found) and the efficiency (i.e. the percentage of the overall time needed to find the final WCET) of the search. The context for the work is that dynamic analysis approaches can deliver WCETs where static analysis is too complex and expensive to apply for the degree of rigour needed by the designers of systems. However our perspective on the problem is that current approaches are too black-box in nature and that using knowledge of the software and hardware (i.e. application specific knowledge) will improve the results.

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. To the best of our knowledge no work, other than our own (Khan and Bate 2009), has considered multi-criteria heuristic functions as part of finding the WCET. The work in (Khan and Bate 2009) has been extended to include more details on the approach significantly more experimental results and analysis of these results. In particular the paper contains a more detailed consideration of the efficiency (i.e. how much processing power is needed) and the reliability of results (i.e. the likelihood of a result being reproduced).

The structure of the paper is as follows. Section 2 surveys issues identified in the literature that affect WCET analysis. Next, Section 3 proposes different methods for WCET analysis using search-based techniques. 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 (2000), amongst others, 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. In this paper it is recognised that it is not practical to deal with all components that might influence the WCET of software, instead the ones generally recognised as being the dominant ones are considered (Wilhelm et al. 2008). The following measures 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 of values that input variables can take.

Groβ (2000) 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 (2000) 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 more difficult. These are widely recognised to be as follows (Wilhelm et al. 2008).

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. 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 (Lunqvist and Stenstrom 1999). 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 (Wilhelm et al. 2008). 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 taken into account in the measures of the actual run-time of the software on the target processor (Bernat et al. 2002). Thus, an execution containing a timing anomaly, when occurring in the test runs, 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 (Wilhelm et al. 2008) 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 (Wilhelm et al. 2008) 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 (Wilhelm et al. 2008).

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) (Jolliffe 2005) 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 could then apply PCA as it could be used in a more focussed manner.

There are four phases to the work:

1. Single criterion—in essence perform WCET analysis as previously demonstrated by Tracey et al. (1998), Wegener (2001, Wegener et al. 1997) and Groβ (2000).
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 Phase 1—Single Criterion Search

A Genetic Algorithm (GA) was developed for automatically generating input test-data. The algorithm was chosen as previous work (Tracey et al. 1998) has used it, albeit 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 (Whitley 1994). Certain common choices and strategies for the algorithm were, however, fixed and used throughout this work 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 calculated. Here reliability is defined as the likelihood of obtaining similar results in practice.

Simplescalar (Burger and Austin 1997) was chosen as the processor simulator on which the working of the input program would be analysed as it is widely recognised as being a stable and accurate simulator, forming the basis for a number of other simulators, e.g. WATTCH (Brooks et al. 2000) and CATS (Kim et al. 2007). The choice of simulator is important as the validity of the data which the search uses is important for the overall results. An inaccurate simulator may give mis-leading results. That said, it is widely recognised that significant cost is involved in using real hardware (Burger and Austin 1997; Tracey et al. 1998). In addition, 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 configurations of Simplescalar and as such have often been used in other work, e.g. (Tan 2006).

- Branch predictor: bimodal
- Branch predictor table size: 2,048 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 (Burger and Austin 1997), 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. (Instructions Cache Accesses refers to the total cache accesses irrespective of whether they are hits or misses.) 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 are chosen so the outputs of their respective analyses tend to give a similar magnitude. In other cases, our intention is to consider different balances between parameters. For example in cases (d) and (e) it is interesting to see what happens what we give data and instruction caches equal weighting and then later look at giving preference to data caches which may show interesting characteristics for software that is more data intensive. Clearly future work could look at more principled methods for deciding these weightings and the factors used, e.g. (Emberson and Bate 2010; Tate and Bate 2010).

(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 (ICA) in the ratio 1:1
(g) BPM, DCM and ICM 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 (Coello 1999). 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 this paper loop iterations not only includes conventional well-structured loops (e.g. *do while* loops) but also less well-structured loops (e.g. due to recursion). The following heuristics were used as the fitness function for this phase. In a similar fashion to the criteria for low-level analysis, the fitness function is calculated across the whole program.

(a) Loop Iterations only
(b) Execution Time and Loop Iterations, combined in the ratio 1:1

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, ICA 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, ICM and LI in the ratio 1:5:5:2
(l) ET, DCM, ICM and LI in the ratio 2:15:5:4
(m) ET, BPM, DCM, ICM and LI in the ratio 1:10:10:10:4
(n) ET, BPM, DCM, 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 (Ermedahl and Gustafsson). The problems comprise a number of programs used to evaluate the performance of different WCET analysis tools. They are widely used in a number of papers and have been the subject of a biennial WCET challenge (Wilhelm et al. 2008). The benchmarks have been categorised by type with the types chosen based on those that are relevant to WCET in terms of their influence from both the final result and problem complexity perspectives. The types considered include a categorisation of whether the software includes features such as nested loops, arrays, recursion or whether the software has no flow dependencies. 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, comprising different input sizes, number of procedure calls, loops (including nested loops) and conditional statements, and a varying degree of dependence on the input data.

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. This is calculated using the average square of the difference between the mean WCET and the WCET from each of the 10 repeated trials. 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.

Table 1 Phase 1 results

Benchmark programs	WCET (mean)	WCET (max)
Factorial	2,053.3	2,188.0
Cover	3,991.0	3,991.0
Insertion Sort (10 inputs) (INS10)	1,328.8	1,333.0
Insertion Sort (50 inputs) (INS50)	17,528.2	18,240.0
Insertion Sort (100 inputs) (INS100)	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

4.1 Phase 1

The results of the initial 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, Fig. 1, of the search show that three programs, *INS100, 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 results in these cases were validated by both checking the program characterisation in (Ermedahl and Gustafsson), e.g. to see whether it should have flow dependencies, and then by manually inspecting the software. For the rest of the paper the eleven benchmarks that are not simple are classed as *interesting benchmarks* and are considered in more detail. The

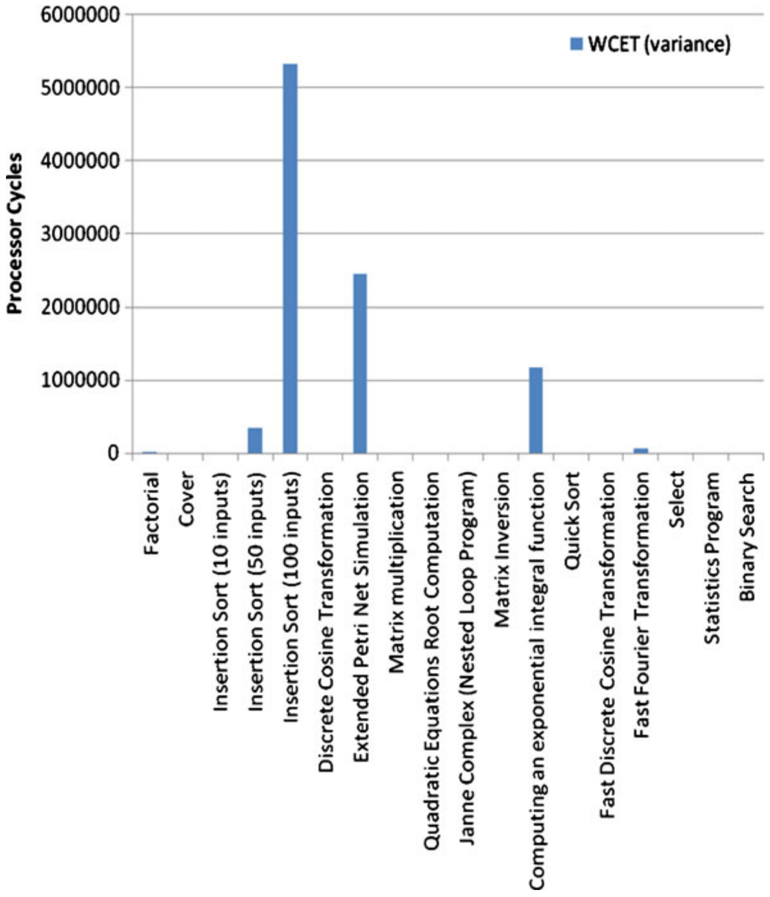

Fig. 1 Reliability results from Phase 1

variances, and thus the reliability, of the other programs varies from under 100 cycles^2 for *INS10* and *Janne Complex* to almost a million cycles^2 for *INS50*. It should be noted that whilst a million cycles^2 may seem large, as this is a variance within units number of clock cycles squared it only corresponds to a standard deviation of a thousand cycles which is just over 5% of the WCET(max) for *INS50*.

4.2 Phase 2

The results of the experiments, for the *interesting benchmarks*, are presented in Figs. 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 and 18, where Phases 2a–2o represent the heuristics (a)–(o) in Section 3.2. The results show that, in general, execution time is a good fitness measure, and finds a high-quality solution in an efficient and reliable manner. For example, Figs. 2, 3 and 4 show the comparative results for the *Extended Petri Net Simulation (PETRI)* program. Throughout the paper, where a "Quality of Solution" figure is presented then for the results of each phase the left hand rectangle represents the average WCET found and the right hand one the maximum WCET across the ten trials.

The results show that, in general a 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 the figures that represent quality of solution for the WCET estimate (i.e. Figs. 2, 5, 7, 9, 10, 12, 13 and 14) produced by Phases 2a, 2b and 2c, the 'WCET (max)' for each of these phases is amongst the lowest values predicted. The figures for reliability (i.e. Figs. 4, 6 and 8—the units for the y-axis of these are clock cycles^2) show that these results also have a large variance, and consequently, low reliability, thereby reducing their usefulness. The speed of producing these results, however, is high as represented in Fig. 3 (the units for the y-axis are seconds of actual compute time on a 2.4 GHz Intel processor). A mean figure across the ten runs is given here. 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 reliability and quality are concerns, then the single criteria of execution time normally offers the best choice.

However, in 7 of the benchmark programs, a multi-criteria fitness measure produced a higher-quality solution than execution time alone. (In the context of this paper where a conclusion is reached that one approach is better or worse than another it is based on manual inspection of the data rather than performing statistical tests. The reason for this approach is in all cases the results give a definitive conclusion.) 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

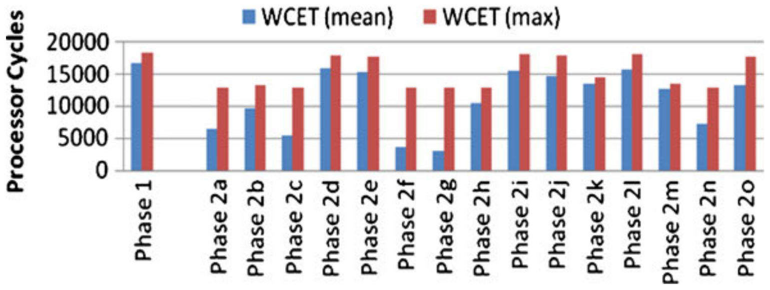

Fig. 2 Extended Petri net simulation (quality of solution)

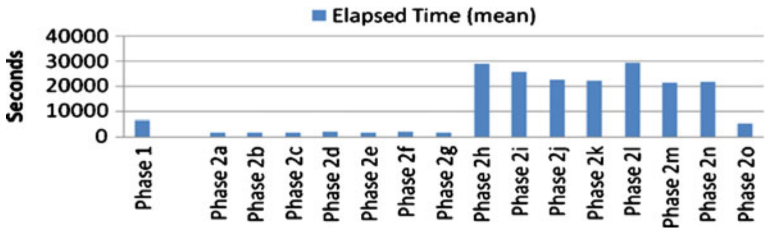

Fig. 3 Extended Petri net simulation (efficiency of solution)

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 Figs. 5 and 6. 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 in Figs. 5 and 6) 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 in Fig. 5). Further, a combination of execution time and data cache misses (Phase 2i in Figs. 5 and 6) 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 that the wrong combination of criteria has a negative effect as shown by the results of Phase 2m for instance.

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 (INS10)* (Figs. 7 and 8), 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 *Insertion Sort program with 50 inputs (INS50)* (Fig. 9), 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 (INS100)* (Figs. 10 and 11), this situation is

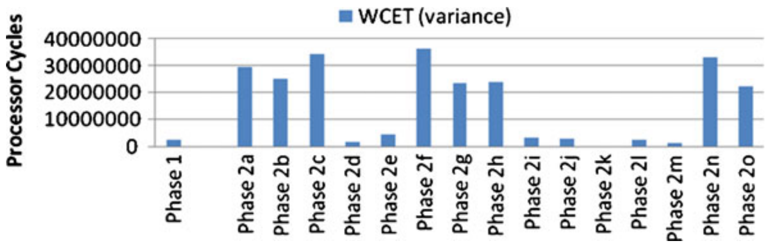

Fig. 4 Extended Petri net simulation (reliability of solution)

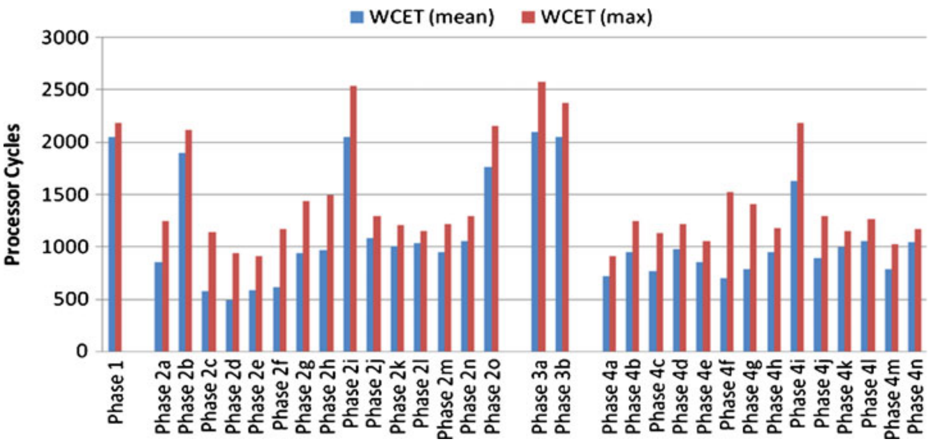

Fig. 5 Factorial (quality of solution)

reversed. In this case, 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 (Phase 2i in Fig. 11) 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, causes branch mispredictions as well. Thus, these two execution measures can be gainfully used for such programs as additional objectives within the fitness function of the genetic algorithm. For example, the *Quadratic Equations Root Computation (QERC)* 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 (Fig. 12) 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

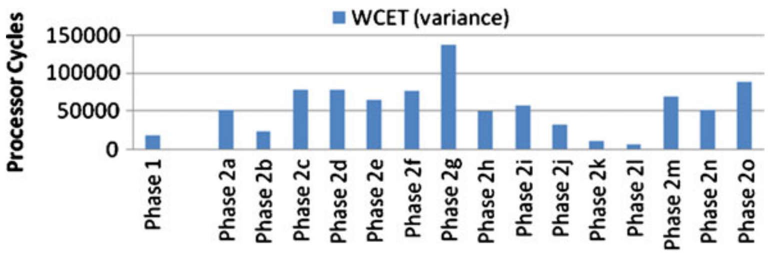

Fig. 6 Factorial (reliability of solution)

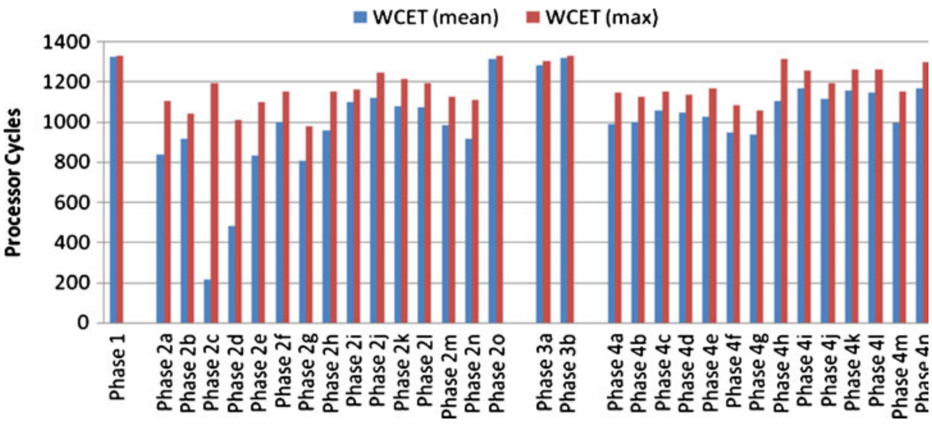

Fig. 7 Insertion sort (10 inputs) (quality of solution)

mispredictions and in particular, instruction cache misses can be gainfully used as additional fitness objectives in programs with similar characteristics.

In some cases, however, the addition of extra criteria confuses the search algorithm. This is particularly true when the objectives 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 the scope of this paper. As a result, the loop will remain unexecuted and the WCET estimate produced may be inaccurate. Figure 13 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 on Fig. 13) 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 bais towards the execution time, as the fitness measure (Phase 2o on Fig. 12) 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 objectives, such as cache misses and branch mispredictions may cause program inputs resulting in only a few iterations of

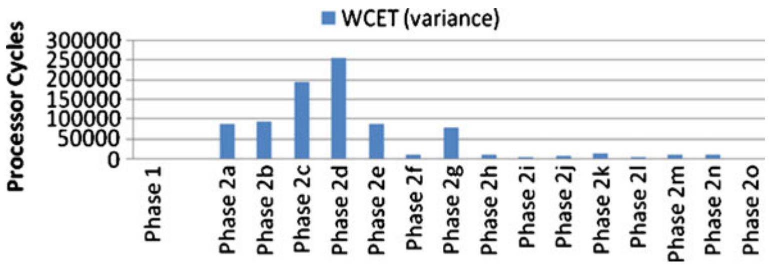

Fig. 8 Insertion sort (10 inputs) (reliability of solution)

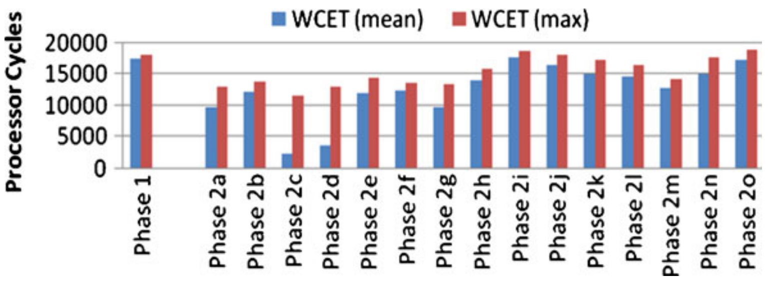

Fig. 9 Insertion sort (50 inputs) (quality of solution)

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.

Another example of this conflict within fitness objectives is seen in the results for the *Binary Search* program (Fig. 14). In this case, only Phase 1 and 2o give the maximum observed execution time showing that execution time alone is a good criterion (in the case of the phase 1 results) but the same results can be found with the right selection of other criteria (in the case of the phase 2o results) within the heuristic. Although this is not as extreme as the previous example, both examples have the potential for a high number of cache misses given the appropriate input test data. The two examples show there is a trend towards better solution for the multi-criteria heuristics when the number of cache misses (particularly instruction cache misses) has a high bias, i.e. when the fitness function is heavily weighted (through the choice of ratios) towards cache misses. The rest of the results almost invariably stop at 238 processor cycles, implying that the fitness assigned by another objective is preventing the genetic algorithm from finding a larger execution time.

Thus, the conflict between objectives 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 objective, or combination of objectives that best guides a search to the program's WCET. Later in the paper this is explored further.

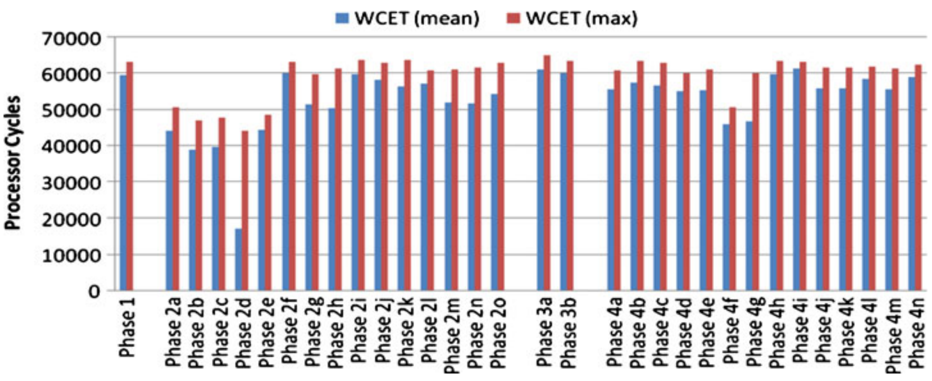

Fig. 10 Insertion sort (100 inputs) (quality of solution)

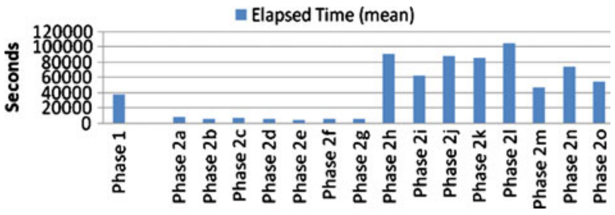

Fig. 11 Insertion sort (100 inputs) (efficiency of solution)

4.3 Phases 3 and 4

The combined results of the experiments for Phases 3 and 4 are presented in this section. Each of these phases is considered in the following sub-sections

4.3.1 Phase 3

The experiment results for Phase 3 show again that execution time is a good fitness measure, in general. Consequently, execution time as the fitness function finds the highest-quality solution in 12 of the 18 benchmark programs. However, in 6 of the 18 benchmark programs, loop counts on its own, or together with execution time as the fitness measure produce the highest-quality solution. This shows that loop counts can gainfully be used to benefit the search. The loop counts measure is particularly useful if there is a single loop within the program, and the number of iterations of this loop varies significantly with the program inputs. For example, in the Factorial program, there are recursive calls to this program, where the number of calls made is dependent on program inputs. Thus, loop counts can be beneficially used as the fitness measure, or as a component of the fitness measure in this search. The results for the Factorial program (Fig. 5) show that loop counts, used as a single fitness measure (Phase 3a on Fig. 5) produces the overall highest-quality solution, while a combination of loop counts and execution time as the fitness function (Phase 3b on Fig. 5), also achieves a better-quality solution than execution time alone.

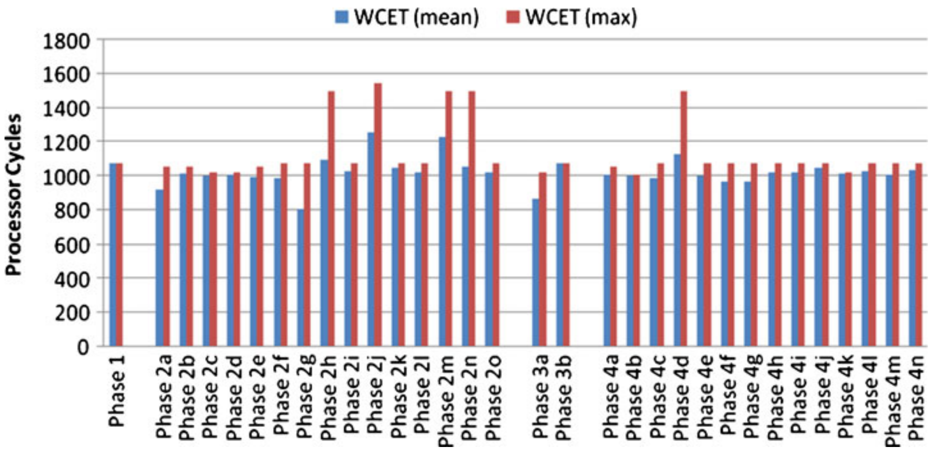

Fig. 12 Quadratic equations root computation (quality of solution)

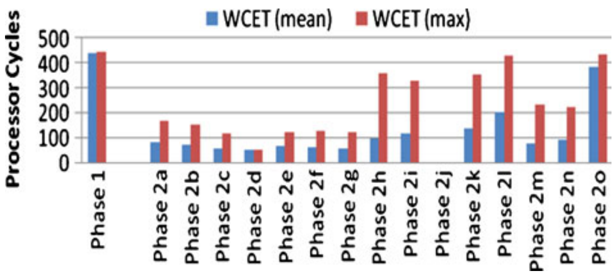

Fig. 13 Janne complex (quality of solution)

Similarly, the Insertion Sort program has a two-level nested loop, where the inner loop iterates an input-data dependent number of times. As the size of the input is increased, the execution time of the program becomes more heavily dependent on the number of iterations of this loop, as it is run a large number of times. Consequently, the number of loop iterations would be a helpful measure in guiding the search in this program, when it is used with a large number of inputs. The results of the Insertion Sort program (Figs. 7, 9 and 10) verify this analysis. Thus, when the program is used with 10 inputs, the solution produced using loop counts as the fitness measure (Phase 3a on Fig. 7) is of a lower-quality than that produced by using execution time as the single fitness measure (Phase 1 on Fig. 7). A combination of execution time and loop iterations as the fitness, however, like execution time used alone, finds the highest-quality solution, but the reliability of finding the highest-quality solution is higher using execution time and loop iterations jointly as the fitness measure (Fig. 15). The results of the Insertion Sort program with 100 inputs (Fig. 10), however, reverse this result as well, as the solution produced using loop iterations alone as the fitness measure has the highest-quality amongst the results for this program. This shows that loop iterations can be successfully used as a measure to guide the search when the size of the program inputs is large, and the values of these inputs significantly alter the number of iterations performed by a single loop, or a set of loops, within the program. Further, maximising loop iterations and execution times simultaneously can increase the reliability of producing the highest-quality solution, particularly in programs with a large number of loops, where loop iterations have a direct correlation with execution times, as a greater number of loop iterations results in a proportionate increase in execution times.

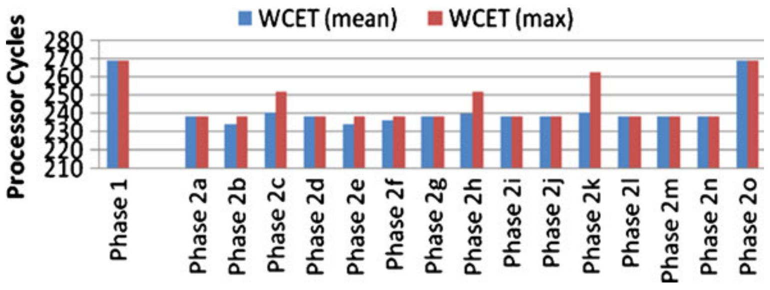

Fig. 14 Binary search (quality of solution)

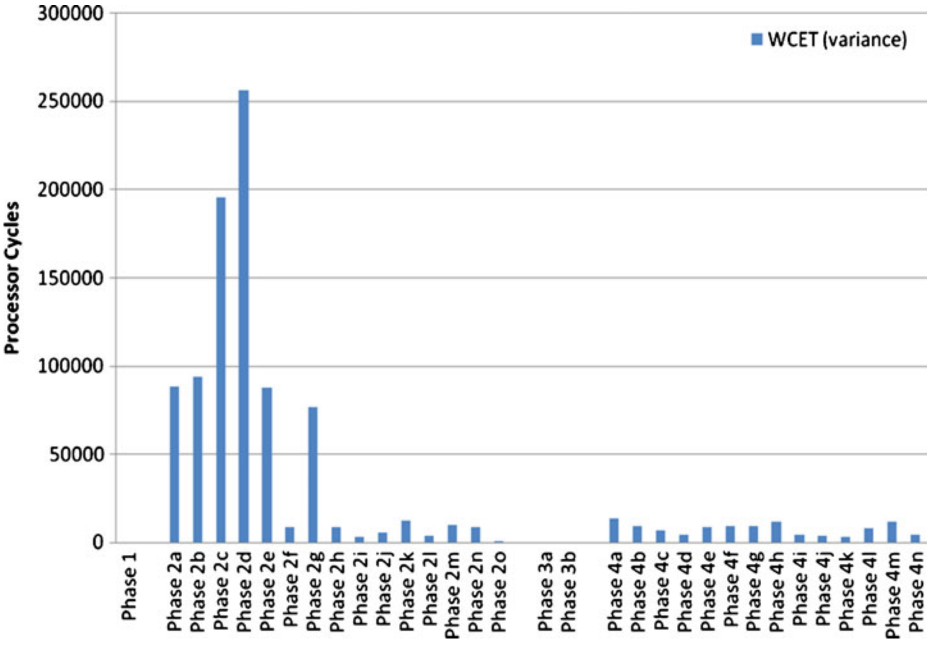

Fig. 15 Insertion sort (10 inputs) (reliability of solution)

4.3.2 Phase 4

The results for Phase 4 show that, in general, combining the criteria does not work well in practice. This was shown by the fact that a heuristic from this phase produced the highest-quality solution in only two of the benchmark programs, the Select and Statistics programs (Figs. 16 and 17, respectively). Further, in both these programs the results from Phase 1, with execution time as the fitness measure, are within 1% of the highest-quality solution, at 99.33% and 99.79% respectively. 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. Thus, combining the low-level and high-level fitness objectives in Phases 2 and 3 does not produce any significant gain in quality, reliability or efficiency.

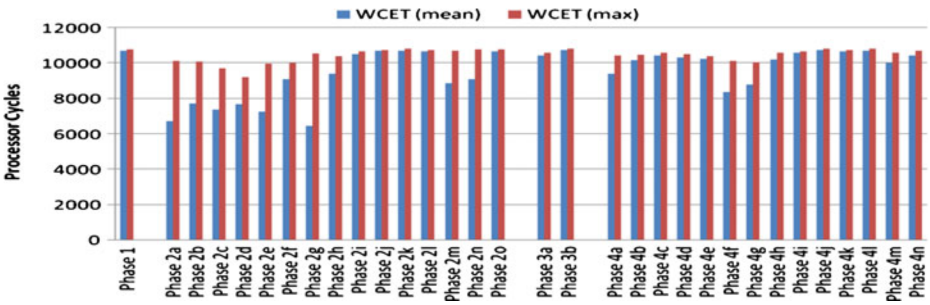

Fig. 16 Select (quality of solution)

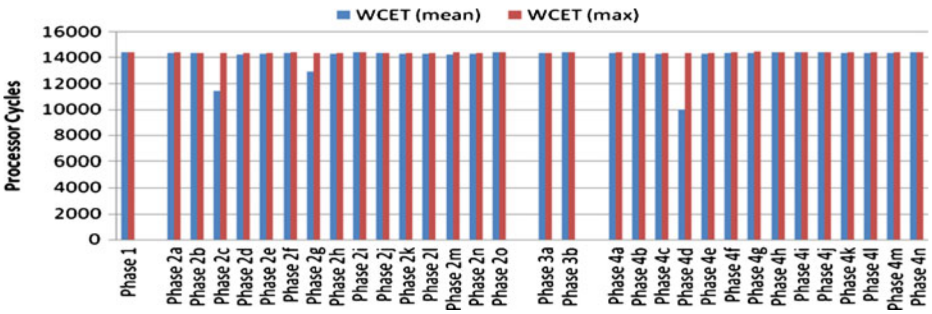

Fig. 17 Statistics program (quality of solution)

However, the results do show that execution time is a good fitness measure as it finds the highest-quality solution in 10 of the 18 benchmark programs. For example, the results for the Fast Fourier Transformation program, in Fig. 18, show that execution time as the fitness measure finds the highest-quality solution which is, additionally, found in an efficient and reliable manner.

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. This suggests that running the genetic algorithm with execution time as a fitness measure for a slightly larger number of generations will find the highest-quality solution. Therefore, there is no significant benefit from using the fitness measure which produced the highest-quality solution, unless it was guaranteed to obtain a solution with better fitness than execution time.

Further, no other fitness measure found as many highest-quality solutions as execution time. The maximum number of highest-quality solutions found by another fitness measure 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 out of the 18 benchmark programs. This shows the general suitability, and applicability, of these fitness measures. However, program analysis should be done beforehand to

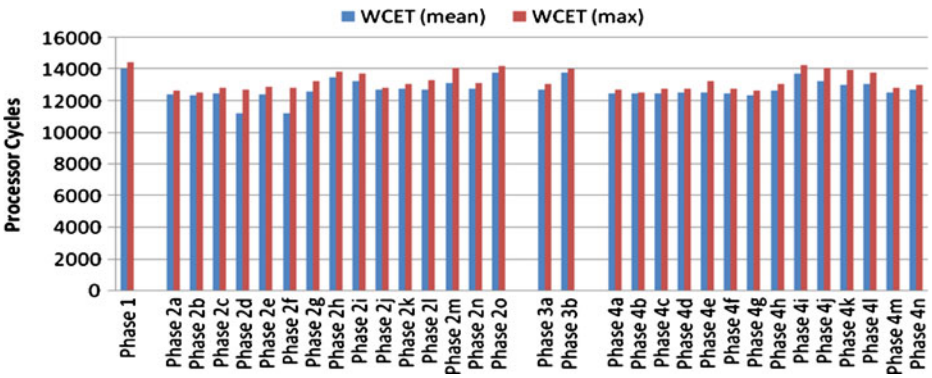

Fig. 18 Fast Fourier transformation (quality of solution)

determine the programs on which use of this particular heuristic as the fitness measure of the genetic algorithm 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 Fig. 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 objectives 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 objectives 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 Figs. 7 and 10, which shows high-quality solutions, with an excellent overall reliability produced by these two fitness heuristics (Phases 3a and 3b in each figure).

In contrast, in a program such as *Quadratic Equations Root Computation (QERC)*, 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 combined presence of many complex loop and conditional statements implies that branch mispredictions or instruction cache misses are likely to have 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. In particular the results for Phase 4(d) are good due to the fact this set of criteria provides an appropriate balance of maximising cache misses (data and instructions) and loop iterations which reflects the dependencies in the software between data and control flow. The results (Fig. 12) 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

Table 2 Testing the proposed heuristics

Benchmark program	FFP	Performance (%)		
		Quality / Accuracy	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

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 entry condition is dependent on the program inputs, then instruction cache misses and

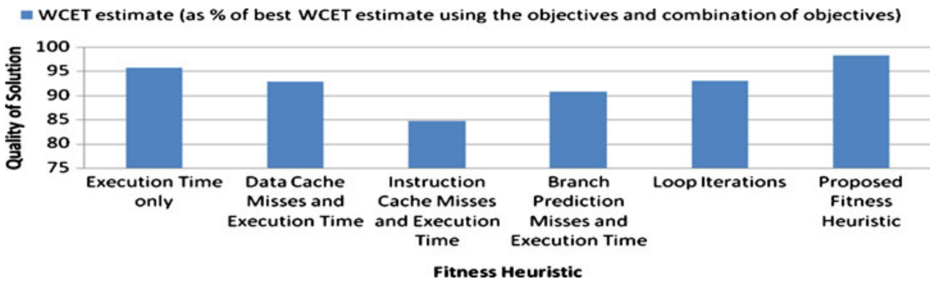

Fig. 19 Comparison of quality of different fitness heuristics

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.

The proposed fitness function for each benchmark problem is listed in Table 2. This table gives the results for the various benchmark programs of 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 performance of the proposed fitness functions for each benchmark problem is summarized in Table 2 and in Fig. 19. 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. 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. The results, especially in Fig. 19, show the proposed fitness functions provide the best results.

5 Conclusions

The paper shows how existing work on search-based WCET analysis can be extended to include criteria other than WCET. However as expected 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

Bernat G, Colin A, Petters S (2002) WCET analysis of probabilistic hard real-time systems. In: Proceedings of 23rd IEEE real-time systems symposium, pp 279–288

Betts A, Bernat G, Kirner R, Puschner P, Wenzel I (2006) WCET coverage for pipelines, technical report for the ARTIST2 network of excellence, August

Brooks D, Tiwari V, Martonosi M (2000) WATTCH: a framework for architectural-level power analysis and optimizations. Proceedings of the conference on 27th international symposium on computer architecture, pp 83–94

Burger D, Austin T (1997) The Simplescalar tool set, version 2.0. SIGARCH Comput Archit News 25 (3):13–25

Burns A, Wellings A (2001) Real-time systems and programming languages, 3rd edn. Addison Wesley, Harlow

Chapman R (1995) Static timing analysis and program proof, PhD thesis, University of York

Coello C (1999) A comprehensive survey of evolutionary-based multiobjective optimization techniques. Knowl Inf Syst 1(3):269–308

Emberson P, Bate I (2010) Stressing search with scenarios for flexible solutions to real-time task allocation problems. Trans Softw Eng, http://doi.ieeecomputersociety.org/10.1109/TSE.2009.58

Ermedahl A, Gustafsson J WCET project / benchmarks. Accessed: 5 May 2008. Available at: www.mrtc.mdh.se/projects/wcet/benchmarks.html

Groβ H (2000) Measuring evolutionary testability of real-time software. Ph.D. thesis, University of Glamorgan/Prifysgol

Harman M (2007) The current state and future of search based software engineering. In: Proceedings of the future of software engineering 2007, pp 342–357

Jolliffe, IT (2005) Principal component analysis, In: Everitt BS, Howell DC (eds) Encyclopedia of Statistics in Behavioral Science, Wiley, New York, 3, pp 1580–1584

Khan U, Bate I (2009) WCET analysis of modern processors using multi-criteria optimisation. Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09), pp 103–112

Kim D, Ha S, Gupta R (2007) CATS: cycle accurate transaction-driven simulation with multiple processor simulators. Proceedings of the Conference on Design, Automation and Test in Europe (DATE), pp 749–754

Kirner R, Puschner P, Wenzel I (2004) 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

Lammermann F, Baresel A, Wegener J (2008) Evaluating evolutionary testability for structure-oriented testing with software measurements. Appl Soft Comput 8(2):1018–1028

Lunqvist T, Stenstrom P (1999) Timing anomalies in dynamically scheduled microprocessors. In: Proceedings of the 20th IEEE real-time systems symposium, pp 12–21

McMinn P (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156

Pohlheim H, Wegener J (1999) Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In: Proceedings of genetic and evolutionary computation conference

Tan L (2006) 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

Tate J, Bate I (2010) Sensornet protocol tuning using principled engineering methods. Comput J, doi:10.1093/comjnl/bxp077

Tracey N, Clark J, Mander K (1998) 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

Wegener J, Mueller F (2001) A comparison of static analysis and evolutionary testing for the verification of timing constraints. Real Time Syst J 21(3):241–268

Wegener J, Sthamer H, Jones B, Eyres D (1997) Testing real-time systems using genetic algorithms. Softw Qual J 6(2):127–135

Whitley D (1994) A genetic algorithm tutorial. Stat Comput 4:65–85

Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenstrom P (2008) The worst-case execution time problem—overview of methods and survey of tools. ACM Trans Embedded Comput Syst 7(3): 1–53

Iain Bate is a lecturer in Real-Time Systems. His research interests include scheduling and timing analysis, design and analysis of safety-critical systems, and engineering of complex systems of systems including sensornets. He is the Editor-in-Chief of the Journal of Systems Architecture and a frequent member of programme committees for distinguished international conferences.

Usman Khan received his MEng (First Class Honours with Distinction) in Computer Systems and Software Engineering from the University of York in 2008. He is a member of the IET and the BCS. His research interests include timing analysis, simulation, real-time systems modelling, control engineering and machine learning.