Towards a Practical WCET Analysis Approach Based on Testing

Thomas Lundqvist
Dept. of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
thomas.lundqvist@chalmers.se

Patrik Sandin
Saab Space AB
SE-405 15 Göteborg, Sweden
patrik.sandin@space.se

***Abstract*—Analyzing the worst-case execution time, the WCET, of a program or task is an important activity when constructing hard real-time systems. Traditional techniques like testing and measurements now face problems due to the introduction of cache memories. This paper presents a new approach that enhances the traditional testing methodology with different analysis methods. The safeness of individual program paths are guaranteed by the use of a safety margin. Furthermore, the approach provides help for the tester to find the critical paths to measure. The approach is demonstrated for a processor containing an instruction cache. The results indicate that this promises to be a simple and practical approach that still can result in low overestimation of the WCET.**

I. Introduction

The determination of the maximum or worst-case execution time, WCET, of a program or task is an important prerequisite when verifying response times in hard real-time systems [1]. Traditionally, the WCET of tasks has been estimated by the use of measurement and testing techniques. By running a program with different inputs while measuring the execution time, the WCET can be estimated. A testing methodology cannot guarantee a safe estimate, i.e., the actual WCET can be underestimated. Nevertheless, measurements can work well in practice since manual inspection of a program often reveal the test cases needed to provoke the worst-case behaviour.

This testing methodology, traditionally used in industry, is now facing problems. Increasingly, microprocessors with cache memories are being introduced in hard real-time systems in order to increase performance by reducing the average memory access latency. One example being the LEON processor core [2], [3], which in its later versions contains both an instruction and a data cache memory. Cache memories introduce new timing dependencies between previously unrelated parts of the program. These dependences are often nonintuitive and make it harder to rely on manual inspection to derive worst-case test cases.

From a testing point of view, cache memories introduce two new sources of uncertainties. The *first uncertainty* is that the execution time of a single program path can vary even when testing with the same input data. The reason is that the number of cache misses depends on the initial state of the cache. This initial state can be hard to control and observe which is fundamental in creating reliable tests. The *second uncertainty* is that the execution of certain program paths can trigger conflict misses that leads to a large nonintuitive increase in the execution time. These program paths might not appear to be interesting from a manual inspection point of view but may still be the paths that cause the longest execution time due to the extra conflict misses.

To restore faith in the testing methodology, we are working on a WCET estimation approach that complements testing with analysis. Our approach is to attack the two sources of cache-related uncertainties by using different analysis methods:

- Single path estimates should be made safe by adding a safety margin to the measured WCET. Ideally, the tightness of the margin (the overestimation) should be controllable by being able to use a range of analysis methods of different complexity. Then, when a large margin might be acceptable, a simple analysis would suffice.
- The tester should be assisted in finding untested dangerous program paths. An analysis should give warning or information about possible cache conflicts to help the tester cover nonintuitive but important program paths.

By eliminating the two sources of uncertainties using relatively simple analysis methods, we hope to restore the same level of confidence in our WCET estimates as we had before cache memories were introduced.

Previous research in WCET analysis [1] has produced a rich variety of analysis approaches. Our approach shares many ideas with other measurement-based approaches [4]–[8]. An important difference however, is that other methods strive for automation in path analysis or test input generation using static analysis methods. Our approach is to instead rely on the tester for assuring program path coverage. This, we believe, will result in a more simple and useful overall approach. Another difference is that many other methods [4], [6], [8] do not include a safety margin to guarantee safe timing analysis leading to potentially unsafe estimates. One notable exception is [5], where they avoid the timing analysis uncertainty by carefully controlling the hardware. In [7], they propose an

This work has been supported by the NRFP (Swedish National Space Board) project 53/07.

approach similar to ours: complementing measurement with analysis to establish that major uncertainties are covered. However, their method relies on more complex probabilistic calculations. We simply use the measured execution time plus a safety margin for obtaining a WCET estimate.

The goal of our approach is to create a range of analysis methods for handling both instruction and data caches. Since this is work in progress, the purpose of this paper is to present the basic ideas. To illustrate these ideas, a simple processor architecture containing an instruction cache will be used. In the next section (Section II), we introduce a small example program to illustrate the two sources of uncertainty when trying to measure the WCET during testing. Then, in Section III and IV we explain how our approach can help to restore confidence in the measured WCET estimates.

II. TESTING AND CACHE MEMORIES

To illustrate the problem with testing we will now only focus on instruction caching and use the example program in Fig. 1. This program consists of a function `a()`, which calls three other functions: `b()`, `c()`, and `big()`. The input data to `a()` is the boolean variables: `x` and `special`, and the boolean vector: `v[]`. These input variables control which program path is going to be executed and thereby which instructions get fetched via the instruction cache.

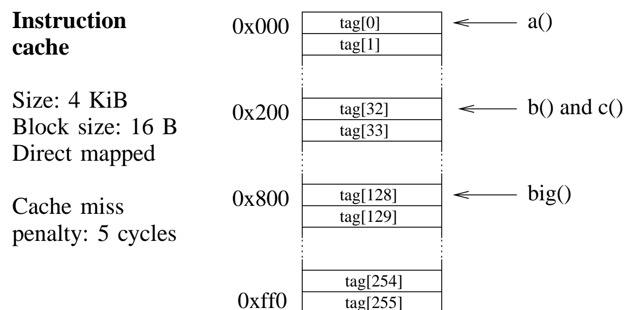
For our example we assume that the program is run on an idealized processor with pipelined instruction execution and a direct-mapped, 16 KiB instruction cache and no data cache. Each machine instruction executes with a constant, fixed latency in the pipeline. A cache miss stalls the execution by a fixed cache miss penalty of 5 clock cycles. All data defining the instruction cache can be found in Fig. 2. This figure also shows how the linker has placed the functions in memory and how big each function is in terms of memory (cache) blocks. An important observation for our example is that the two functions `b()` and `c()` map to the same location in the instruction cache. This is a potential source of cache conflict misses as we will see later.

We will now look at what happens when we measure the execution time of this program by testing different input data. Table I shows how the real execution time varies depending on the input we use. For example, test case 1 makes the program call function `b()` before the loop is entered as well as inside each loop iteration. The real execution time varies between 4140 clock cycles and 4210 clock cycles due to *cold misses* in the instruction cache. If the cache already contains the needed memory blocks in the beginning of the execution we get 4140 clock cycles. If the cache is empty, we get 4210 clock cycles.

Table I also illustrates how a tester might proceed when trying to estimate the WCET of function `a()`. We assume that during and between measurements we have no control over the cache content. Our measurements will therefore end up anywhere in the range given by the real execution time. For the WCET estimation, the loop is the natural starting point since programs often spend most of their time in loops. The first two test cases, 1 and 2, cover the two different alternatives

```
1  void a(x, v[], special)
2     if (x)
3        b()
4     else
5        c()
6     if (not special)
7        for (i = 0 ; i < N ; i++)
8           if (v[i])
9              b()
10          else
11             c()
12    else
13       big()
```

Fig. 1. The example C program. The function `a()` is shown. This function calls three other functions: `b()`, `c()`, and `big()`. The boolean input variables `x`, `v[]`, and `special` control the execution path.



Function	Address	Cache address	# of blocks
`a()`	0x0000-0x009f	0x000	10
`b()`	0x0200-0x023f	0x200	4
`c()`	0x1200-0x124f	0x200	5
`big()`	0x1800-0x1bff	0x800	64

Fig. 2. Instruction cache configuration and placement of functions. Each block in the cache has a tag identifying the memory block currently cached. The functions map to different locations in the cache depending on their memory address. The blocks from `b()` and `c()` might conflict in the cache.

inside the loop: calling `b()` or calling `c()`. The tester would find that `c()` has a longer execution time and would maybe continue with the additional test cases, 3 and 4, to cover the other obvious program paths. The final estimate of the WCET would become 5200 clock cycles (test case 3).

In this testing example, the WCET was underestimated due to two reasons. First, we did not know how the real execution time varies for the program paths we measured and even if we run the same test case multiple times, we cannot know if we

TABLE I
TEST-CASES USED WHEN ESTIMATING THE WCET OF THE EXAMPLE PROGRAM IN FIG. 1. THE NUMBER OF LOOP ITERATIONS IS $N = 100$.

Test case	`x`	`v[]`	`special`	Real execution time	Measurement
1	true	true $\forall i$	false	4140–4210	4154
2	true	false $\forall i$	false	5160–5235	5177
3	false	false $\forall i$	false	5150–5225	5200
4	false	false $\forall i$	true	650–1045	655
(5)	false	alternating true–false	false	6650–6725	

have covered the whole range. For example, among the four paths tested, the estimated WCET should have been 5235 not 5200. The other reason for underestimating the WCET is that we failed to test the dangerous program path represented by test case 5 in Table I. This test case causes `b()` and `c()` to be called in an alternating way so that 4 extra *conflict misses* occur in each iteration. Testing this path would have given an estimate closer to the real WCET of 6725 clock cycles.

III. Safety margin methods for instruction caches

In the previous section we found that the WCET was underestimated due to two reasons. First, the execution time of a single path could vary due to the undefined initial cache content. The second reason was that a critical program path was not tested. We will now see on how our approach can handle these problems. In this section, we will take a look on methods to handle the varying execution times. In the next section (Section IV), we will present methods to help with the second problem, finding critical untested paths.

To obtain a safe WCET estimate for a single program path despite the variations possible due to the undefined initial cache state, we add a safety margin to our measurements. This safety margin should ideally be as small as possible to reduce the overestimation. Still, we want to have a range of methods available so that less complex methods can be used when some overestimation can be tolerated. This section presents three such methods: the constant bound method, and the dynamic and static cache footprint methods. We demonstrate these methods using the example program and the direct-mapped example system from Section II. However, the same approach also handles set-associative instruction caches with LRU (Least Recently Used) replacement.

To be able to calculate a safety margin we need to be able to reason about how the initial cache state can influence the future execution time. An important requirement on the processor architecture is that the effect of a change in the initial cache state has a constant, fixed penalty on the future execution time. Another way of expressing this is that a change in the initial timing state in the system has a bounded timing effect [9]. In our example system this requirement is fulfilled. For example, we know that an invalidation of a cache block can cause at most 5 clock cycles penalty on the future execution time. This makes it possible to calculate a safety margin based on the number of initially undefined cache blocks as:

$$\text{margin} = B * P$$

where B is an upper bound on the number of undefined initial cache blocks and P is the cache miss penalty. The worst-case assumption here is that during measurements, the cache might contain useful blocks so that fewer cold misses occur compared to the possible run-time behaviour. The safety margin compensates for this risk.

To calculate the margin we need to find an upper bound on the number of undefined cache blocks B. We now present three different methods to estimate this upper bound.

A. The constant bound method

The constant bound method represents the most simple approach. Here, we simply assume that all cache blocks in the cache are undefined and potentially used by the program:

$$B = \text{total number of cache blocks}$$

For our example in Section II we get $B = 256$ and a margin of $256 \times 5 = 1280$ clock cycles.

If the overestimation can be tolerated, this method has important advantages. The margin calculated is *program independent*. Thus, no analysis of the program is needed. Also, for programs larger than the cache size, this method gives the best estimate.

B. Dynamic cache footprint method

The next method, the dynamic cache footprint method, has the potential of reducing the overestimation by limiting B to the actual number of blocks touched when executing a certain program path. This method relies on collecting instruction fetch trace data during testing and requires hardware or simulator support. Mapping trace data to memory locations then reveals how many cache blocks that are touched for a certain program path:

$$B(p) = \text{touched blocks for path } p$$

This results in the lowest possible overestimation since it calculates an individual safety margin for each measured program path (test case). For example, for test case 5 in our previous example we would find that the number of touched blocks is $B(5) = 15$ and the margin would become $15x5 = 75$ clock cycles, exactly covering the real variation in execution time.

C. Static cache footprint method

The static cache footprint method simplifies the dynamic version by relying on information from the linker instead. Knowing the placement and size of functions in memory, the total number of blocks occupied by the program can be calculated:

$$B = \text{total program footprint in cache}$$

This will typically produce a bound B larger than what the dynamic method does. For our example, $B = 79$, and the margin is $79 \times 5 = 395$ clock cycles. One advantage is that the same margin can be used for all measured paths.

D. Discussion

The possible WCET overestimation resulting from the different safety margin methods is listed in Table II. The overestimation varies since the measured execution times varies. Thus, we could do multiple measurements and pick the lowest one to reduce the overestimation. For our example program, the dynamic cache footprint method gives the lowest overestimation. However, if we increase the number of loop iterations to $N = 1000$, all methods result in low overestimation.

TABLE II
THE WCET OVERESTIMATION BY THE DIFFERENT SAFETY MARGIN METHODS FOR TWO DIFFERENT VALUES OF N, THE NUMBER OF LOOP ITERATIONS IN THE EXAMPLE PROGRAM.

Method	Margin	Overestimation $N = 100$	$N = 1000$
Constant bound	1280	17.9%–19.0%	1.8%–1.9%
Dynamic cache footprint	75	0.0%–1.1%	0.0%–0.1%
Static cache footprint	395	4.8%–5.9%	0.5%–0.6%

The safety-margin methods can also be used in combination with hardware control strategies. For example, by adopting some of the techniques mentioned in [5], like cache flushing or locking, the execution time variation can be completely or partly eliminated. This would also reduce the safety margin thus allowing for a trade-off between hardware control, analysis complexity, and overestimation.

IV. TEST COVERAGE WARNING METHODS FOR INSTRUCTION CACHES

The safety margin methods only guarantees a safe WCET estimate for individual program paths. For path analysis, we rely on the tester to provide sufficient coverage. This can be easy if the program only contains a single path. However, given multiple paths, we want to give the tester information or warnings about possible conflict misses in the program to help the tester cover dangerous paths.

The approach taken is to rely on the linker to provide placement and size information about all functions in the program. The program memory regions can then simply be mapped to cache regions to indentify conflicting areas. For example, in our previous example, such an analysis would quickly reveal that 4 memory blocks in `b()` and `c()` map to conflicting regions.

Given information about conflicting regions in the code, the tester proceeds to assess the risk:

- If a path exists through the program that passes through conflicting regions in a repeatable and alternating way, that path should be tested. Alternating between two different regions could trigger conflict misses and if these regions are inside a loop it could cause a large increase in execution time.
- When inspecting different paths, the worst-case path found so far should be prioritized to see if it conflicts with some other region. Conflicts with the worst-case path have a great potential of causing an increase in the WCET.

Following these steps, a tester should have had no difficulties in finding test case 5 in our previous example.

Given the information about conflicts, there is also another important option. Since conflict misses are unwanted also when striving for good average performance, it can be of interest to control the linking phase in order to avoid conflicts. This can be difficult for a direct-mapped cache but it can be an important option for set-associative caches.

V. DISCUSSION AND FUTURE WORK

Our approach is to let the tester be responsible for finding critical paths and thereby obtaining safe WCET estimates. This simplifies the analysis needed and should work well for programs with few program paths to test. However, further studies are needed of more realistic programs and benchmarks to assess the general applicability of this approach.

In the previous sections we have demonstrated our approach for direct-mapped instruction caches. However, this study is part of an ongoing project that aims to develop a full set of methods to handle WCET analysis for the LEON processor core [2], [3]. Thus, we need to handle both instruction and data caches, which also are set-associative with LRU replacement. Apart from the WCET analysis we should also be able to estimate the effect of caches when doing a response-time analysis of tasks that use preemptive scheduling.

Another important goal is to be able to support regression testing. Here, we believe a successful approach will be to support the tester with information about the impact of program changes by highlighting differences in margin calculations and conflicting program regions between test runs.

VI. CONCLUSION

In this paper we have presented ideas for a new approach to WCET analysis. Based on testing and the measuring of critical program paths we add a safety margin to the measured execution times to obtain safe WCET estimates. The methods presented are fairly simple but can still lead to low overestimation of the WCET. Further studies are needed to confirm the general applicability of the approach.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, Apr. 2008.

[2] ESA Microelectronics, "LEON2-FT IP core," http://www.esa.int/TEC/Microelectronics.

[3] Gaisler Research, "LEON processor cores," http://www.gaisler.com.

[4] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based worst-case execution time analysis," in *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2005. SEUS 2005*, 2005, pp. 7–10.

[5] J.-F. Deverge and I. Puaut, "Safe measurement-based wcet estimation," in *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, R. Wilhelm, Ed., Dagstuhl, Germany, 2007.

[6] G. Bernat, A. Colin, and S. M. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Proceedings of the 23th IEEE Real-Time Systems Symposium (RTSS'02)*, 2002, p. 279.

[7] S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both?" in *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, C. Rochange, Ed., Schloss Dagstuhl, Germany, 2007.

[8] M. Lindgren, H. Hansson, and H. Thane, "Using measurements to derive the worst-case execution time," in *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA'00)*, 2000, p. 15.

[9] T. Lundqvist, "A wcet analysis method for pipelined microprocessors with cache memories," Ph.D. dissertation, Chalmers University of Technology, Göteborg, Sweden, June 2002.