Avoiding the WCET Overestimation on LRU Instruction Cache*

L. C. Aparicio†, J. Segarra†, C. Rodríguez‡, J. L. Villarroel† and V. Viñals†
†{luisapa, jsegarra, jlvilla, victor}@unizar.es, DIIS, Universidad de Zaragoza
‡acprolac@si.ehu.es, DATC, Universidad del País Vasco

Abstract

The WCET computation is one of the main challenges in hard real-time systems, since all further analysis is based on this value. The complexity of this problem leads existing analysis methods to compute WCET bounds instead of the exact WCET. In this work we propose a technique to compute the exact instruction fetch contribution to the WCET (IFC-WCET) in presence of a LRU instruction cache. We prove that an exact computation does not need to analyze the full exponential number of possible execution paths, but only a bounded subset of them. In the benchmark codes we have studied, the IFC-WCET is up to 62% *lower than a bound computed with a widely used approach, and the difference between the number of possible execution paths and the ones relevant for the analysis is extremely large.*

1 Introduction

Computation of the Worst Case Execution Time (WCET) of a task is one of the main challenges in the study of hard real-time systems. WCET is difficult to obtain since it depends on both hardware and software, but it is needed in order to guarantee the time requirements of the system.

Given hardware components with a fixed latency, the WCET can be computed from the partial WCET of each basic block of a program. However, in order to improve performance, current processors perform many operations with a variable duration. This is mainly due to speculation (control or data) or to the use of hardware components with variable latency. Branch or store-load independence predictors fall in the first category, whereas memory hierarchy and datapath pipelining belongs to the second one. A memory hierarchy made up of one or more cache levels takes profit of program locality and saves execution time and energy consumption by delivering data and instructions with an average latency of a few processor cycles. Unfortunately, the cache behavior depends on the past references and, in order to compute in advance the latency of a given access, it is required to know the previous access sequence. So, to compute the exact instruction fetch contribution to the WCET (IFC-WCET) in presence of caches it is needed to analyze each and every execution path.

In the literature we find different methods to approach the WCET problem in presence of caches [11]. Due to the difficulty of the problem, most of these methods try to solve it by dividing the problem in two (or more) simpler steps [2, 3, 4, 5, 6, 7, 10]. In the first step, they avoid the combinatorial explosion of paths by not always remembering the whole history of the followed one. This is generally referenced as *path merging*, where several different possible executions reaching a concrete point are merged, guaranteeing that the resulting combined path is not better than any of the original ones. Having less (or even just one) paths to analyze, it is possible to classify each memory access either as a hit, a miss or an uncertainty, which may also have additional information (e.g. A *first miss* classification in *Static Cache Simulation* means a miss followed by successive hits [7]). Since WCET bounds must be upper bounds, any uncertainty must be considered as the worst possible case. In a second step, the timing of executing each instruction is considered to compute the WCET bound.

In this paper we propose a technique aimed towards the exact computation of the IFC-WCET in presence of an instruction cache. We prove that an exact IFC-WCET computation does not need to analyze the full exponential number of possible execution paths, but only a bounded subset of them. Our approach allows to discard most of them and focus just on the relevant ones. In the benchmark codes we have studied, the IFC-WCET is up to 62% lower than a bound obtained by the well-known *Static Cache Simulation* method [7]. Also, the number of execution paths that we need to analyze may be up to $2\,800$ orders of magnitude lower than the possible execution paths.

This paper is structured as follows. In Section 2 we bound the number of paths to be analyzed. In Section 3 we show our obtained results. Finally, conclusions and future work are presented in Section 4.

*This work has been supported by the project Grupo Consolidado de Investigación of the Diputación General de Aragón and by the project TIN2007-66423 of the MEC/MCyT of Spain.

2 Bounding the Number of Relevant Paths

In this section we prove that in order to obtain the exact instruction fetch contribution to the WCET, the number of paths to explore is bounded and much lower than the number of possible execution paths.

We assume that the timing effect of instruction cache is independent of any other hardware component. We consider a common replacement policy: least recently used (LRU), where cache content is ordered by the time of the last reference. Having an LRU replacement policy, in order to be general, we are going to deal with the most challenging LRU cache architecture for the WCET analysis, which is a fully associative cache (no conflict misses) with unlimited size (no capacity misses). This avoids all evictions and tracks a global ordering of blocks (instead of a per-set ordering), which maximizes differences in cache states.

2.1 Bounds inside Loops

We are going to prove that an n-iteration loop enclosing p-alternative paths can only lead to $\sum_{i=1}^{min(p,n)} \frac{p!}{(p-i)!}$ different cache states (Proposition 3), and not to p^n, which is the number of distinct paths in the loop. For now on, we will assume that the number of loop iterations n is bigger than the number of alternative paths p inside the loop, which is the usual case and results in cleaner formulas. $\sum_{i=1}^{p} \frac{p!}{(p-i)!}$ is independent on n and, for practical purposes, nicely bounded. In order to bound the maximum number of different cache states we give the following definitions, properties and propositions about *sequences of instruction block addresses*. An instruction block address is the address issued by the processor to fetch an instruction, without the least b significant bits for an instruction cache organized in blocks of 2^b bytes (also called lines in the cache literature). To shorten, from now on a sequence of instruction block addresses will be simply called an *address sequence*.

Definition 1. *We define the Ordered Set of Blocks (OSB) as the set of different blocks ordered in increasing order of access time in an address sequence. That is, let A be the address sequence generated along an execution path of a program. OSB_A is the ordered set of different instruction blocks fetched during the execution of such a path. Note that the order of a block in an OSB depends only on the time of its last access (LRU ordering).*

For example, the OSB associated with the address sequence $[b_i, b_i, b_j, b_j, b_k, b_k]$ is $\{b_i, b_j, b_k\}$, and in another example, the OSB associated with the address sequence $[b_i, b_i, b_j, b_j, b_k, b_k, b_j, b_j]$ is $\{b_i, b_k, b_j\}$.

For notation purposes, we denote OSB_{AB} as $OSB_A \cup OSB_B$ and we assume $A \neq B$. OSBs are ordered sets, therefore the following properties are trivially deduced by the general theory of sets:

Property 1. $OSB_A \bigcup OSB_A = OSB_A$

Property 2. $OSB_A \bigcup OSB_B \neq OSB_B \bigcup OSB_A$

Property 3. $OSB_A \bigcup OSB_B \bigcup OSB_A = OSB_B \bigcup OSB_A$

Definition 2. *(Cache State) Let A be the address sequence associated with an execution path of a program. $CS(A)$ represents the cache state that is reached after executing the address sequence A.*

Definition 3. *Let OSB_A be the OSB associated with the address sequence A. $CS(OSB_A)$ represents the cache state that is reached after the ordered access to the blocks belonging to OSB_A.*

Proposition 1. $CS(A) = CS(OSB_A)$

Let A be the address sequence associated to an execution path of a program, and OSB_A its associated ordered set of blocks. $CS(A)$ is equal to $CS(OSB_A)$.

Proof. This proposition is trivially deduced from definitions 1, 2 and 3: by construction, OSB_A is the LRU-filtered version of A, and cache states store an LRU-filtered version of a given address sequence. □

Proposition 2. $CS(A, B) = CS(OSB_{AB})$

Let A, B be two different address sequences. The cache state that is reached after executing first the address sequence A and later the address sequence B is equal to $CS(OSB_{AB})$

Proof. $CS(A, B) = CS(OSB_A \bigcup OSB_B) = CS(OSB_{AB})$ □

Rule 1. $CS(A, A) = CS(A)$

The cache state reached after executing an address sequence A twice consecutively, is equal to the cache state that is reached when this sequence is executed only once.

Proof. $CS(A, A) = CS(OSB_{AA}) = CS(OSB_A) = CS(A)$ □

Rule 2. $CS(A, \cdots_{(n)} \cdots, A) = CS(A)$

The cache state reached after executing n times an address sequence A is equal to the cache state that is reached when this sequence is executed only once.

Proof. By induction, applying Rule 1. □

Rule 3. $CS(A, \cdots, A, B, \cdots, B) = CS(A, B)$

The cache state reached after executing n times the address sequence A, followed by the execution of the address sequence B m times is equal to the cache state reached after executing the sequence A once and then B once.

Proof. $CS(A,\cdots,A,B,\cdots,B) = CS(OSB_{A\cdots AB\cdots B}) = CS(OSB_{AB}) = CS(A,B)$ □

Rule 4. $CS(A,\cdots,A,B,\cdots,B,A,\cdots,A) = CS(B,A)$

The cache state reached after executing i times the address sequence A, followed by executing j times the address sequence B, and followed by executing k times the address sequence A is equal to the cache state that is reached after executing the sequence B and later the sequence A.

Proof. $CS(A,\cdots,A,B,\cdots,B,A,\cdots,A) = CS(OSB_{A^{\cdots(i)}\cdots AB\cdots(j)\cdots BA\cdots(k)\cdots A}) = CS(OSB_{ABA}) = CS(OSB_{BA}) = CS(B,A)$ □

Proposition 3. *Let L be an n-iteration loop containing p different instruction paths inside. The maximum number of states in the instruction cache at the ending of L is $\sum_{i=1}^{\min(p,n)} P_p^i$ where P_p^i are the permutations of i elements selected from p elements, that is $\sum_{i=1}^{\min(p,n)} \frac{p!}{(p-i)!}$.*

Proof. Let A_k, with $k = 1,\ldots,p$ be the address sequence associated with the k-th alternative path inside the loop.

If we always take the same path k, applying Rule 2, there is only one different CS ($CS(A_k)$). Since there are p alternative paths we have p cache states ($CS(A_1)$,$CS(A_2)$,...,$CS(A_p)$), that is, P_p^1.

If we always take exactly two paths i, j ($i \neq j$), applying Rules 3 and 4, we have two cache states ($CS(A_i, A_j)$ and $CS(A_j, A_i)$). Thus, accounting all possible pairs of different paths we have P_p^2 cache states. This calculation can be repeated until considering that all p paths can be taken, which results in P_p^p cache states. However note that, if the number of iterations n is lower than p, one could never traverse p different paths, but only n.

Accounting all previous cases we have that the number of different cache states is $\sum_{i=1}^{\min(p,n)} P_p^i = \sum_{i=1}^{\min(p,n)} \frac{p!}{(p-i)!}$. □

Figure 1 shows a comparative between the number of execution paths and the number of different cache states according to the number of iterations in a loop enclosing 2, 4 and 16 alternative paths. Note that the y-axis is logarithmic. Although the number of different cache states initially grows exponentially, it is always below the number of different paths, and remains constant when the bound is reached.

Corollary 1. *In order to compute the exact IFC-WCET, every time we reach an instruction shared by several execution paths (e.g. at a loop end) we can select, for each distinct cache state, the path with the longest cumulative execution time, discarding the others. From this pruning point on, analysis (instruction cache state and cumulative execution time tracking) must take place separately on each selected path, until another pruning point is reached.*

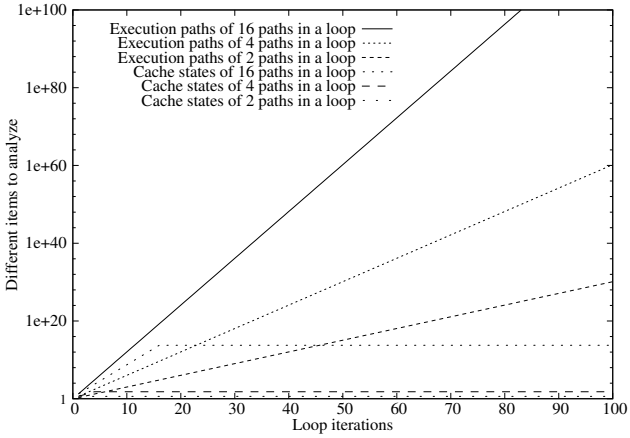

Figure 1. Number of execution paths and instruction cache states according to the number of iterations in a loop.

Corollary 2. *For any value of the number of paths p and loop iterations n in a loop, the number of different cache states that can be reached at the end of the loop is lower than $\sum_{i=1}^{\infty} \frac{1}{i!} \times p! = e \times p!$. Additionally, this maximum number of different cache states is reached with $\lceil log_p(e \times p!) \rceil$ iterations.*

Corollary 3. *In an s-way set-associative LRU cache, the maximum number of cache states is also bounded to $\sum_{i=1}^{p} P_p^i$, i.e. the number of sets does not affect the cache states bound. This can be easily seen by taking into account that the worst case in a set-associative LRU cache is achieved when all accesses are mapped to the same set. This case is the same as having a totally associative cache of smaller size, so its bound remains the same than that of a totally associative cache.*

Corollary 4. *The previous results can be directly applied on nested loops. Let us have an outer loop $L1$ with p^{L1} alternative paths inside, one of them p_k^{L1} containing an inner loop $L2$ with p^{L2} alternative paths. The path p_k^{L1} actually accounts for $\sum_{i=1}^{p^{L2}} P_{p^{L2}}^i$ alternative paths in terms of calculating the number of instruction cache states in loop $L1$. This can be extended for any nesting level.*

Corollary 5. *When a loop L may be entered by n different instruction cache states, the maximum number of states in the instruction cache at the end of L is $n \times \sum_{i=1}^{p} P_p^i$. Note that this bound is not cummulative with Corollary 4, which already accounts these cases in nested loops.*

2.2 Bounds outside Loops

Code outside of loops is generally not a problem, since the number of traversed branches is much more limited. In

any case, a branch can be considered as a branch inside a loop with just one iteration.

On the other hand, it is highly frequent that different execution paths end up traversing the same basic block after convergence instructions. In these cases, cache states tend to converge for any replacement policy [9]. Thus, any application of Corollary 1 several instructions after a convergence point should reduce the number of cache states. This is also applicable to the common instructions after ending a loop, i.e. the bound on the number of cache states given by Proposition 3 essentially affects the analysis of the loop, but tends to 1 after analyzing subsequent instructions.

Finally, take into account that our proposal follows the execution flow, and thus functions are analyzed each time they are called. This means that analyzing function calls needs no special treatment, even for recursive ones, since the *call* and *return* statements are unconditional jumps.

These considerations cover any well-structured code.

3 Results

As a proof of concept we have developed a tool which follows (without actually executing) the program control flow and computes the exact IFC-WCET and the overall WCET under simplified hardware assumptions. Following the control flow is accomplished by simply decoding instructions sequentially and following unconditional branches. For conditional branches, annotations (e.g. number of iterations in a loop) must be followed when they exist. If there is no annotation in a conditional branch, analysis is forked. This means that whenever an non-annotated conditional branch is reached, our tool takes both paths, making a separate analysis for each one. Obviously, this leads to a combinatorial explosion, which is solved by applying Corollary 1. The application of this corollary essentially means that every path analysis must stop at a predefined pruning point. When every single path analysis reaches this point, cache states are compared. If two paths have an identical state, the path with the lowest cumulative WCET until that point can be pruned.

Our tool models a simple processor which processes instructions in two sequential non-pipelined stages: instruction fetch and execution. Instruction fetch takes 1 or 60 cycles depending on whether it hits or misses in the instruction cache. Non-memory instructions finish their execution in the next cycle. Memory instructions (load and store) spend in their execution phase 60 cycles, since we always assume a data cache miss.[1] For our experiments we have used a subset of those found in [1] plus some other interesting/alternative benchmarks, and they are commonplace in WCET studies [4, 6, 7, 8, 10]. We have selected those that include conditional sentences inside loops, have bounded iterations in loops and do not contain unstructured code. Benchmarks too simple or too similar to those selected have not been considered. We test 2-way set associative caches with varying sizes of 128, 256 and 512 bytes, and block sizes of 8, 16 and 32 bytes. These small cache sizes are reasonable for the selected benchmarks.

[1] Experiments performed assuming other miss penalties and other data cache behavior, such as hit-always, may change the worst case path, but the trends observed in IFC-WCET remain.

3.1 Exact IFC-WCET vs. SCS Bound

All events consuming time contribute to the WCET and determine the worst path. Since the focus of this paper is the instruction fetch analysis, we isolate the IFC-WCET from the overall WCET by subtracting the time contribution of both data access and execution. In order to get a clear comparison, the method to compare with must be also specialized on instruction cache analysis and not device-specific. We have chosen the *Static Cache Simulation (SCS)* method [7], which is well established and documented.

Figure 2 shows the exact IFC-WCET normalized to the IFC-WCET bound obtained by SCS. Additionally, the figures also show the instruction hit ratio for the worst-case execution path. Given that our IFC-WCET is exact, it must be always equal or lower than the SCS bound. Reductions in exact IFC-WCET appear when SCS is not able to accurately classify an instruction fetch into one of its characterizations types (*hit*/*miss*/*first-hit*/*first-miss* by loop level). We can highlight two interesting facts from the figure: 1) The approximation quality of *SCS* does not correlate with cache size (e.g. *qurt*-16B across all cache sizes) nor block size (e.g. in *bubble*-128B the *SCS* approximation improves quality as the block size increases, whereas in *qurt*-256B it happens the contrary). 2) The instruction hit ratio for the worst-case execution path is not a good indicator to guess *SCS* accuracy. See for instance *bubble*-128B, where the hit ratio remains flat across all the block sizes, but SCS accuracy increases significantly. The reason why there is no correlation between the architectural parameters and the IFC-WCET bound accuracy is that in every experiment (cache and block size), the worst case path can differ. So, the importance of classifying correctly a given instruction fetch as hit or miss depends critically on the location of the instruction: an overestimation made to an instruction belonging to the worst-case path may be amplified, for instance by an enclosing loop. Additionally, a misclassification can even change the worst case path itself. Both possibilities happen in the analyzed benchmarks.

Regarding *qurt* benchmark, the IFC-WCET reduction found by the exact method is significant and sustained across all cache configurations (up to 62%). This benchmark is much bigger than the others, which means that it does not even fit in the biggest cache. Thus, depending on

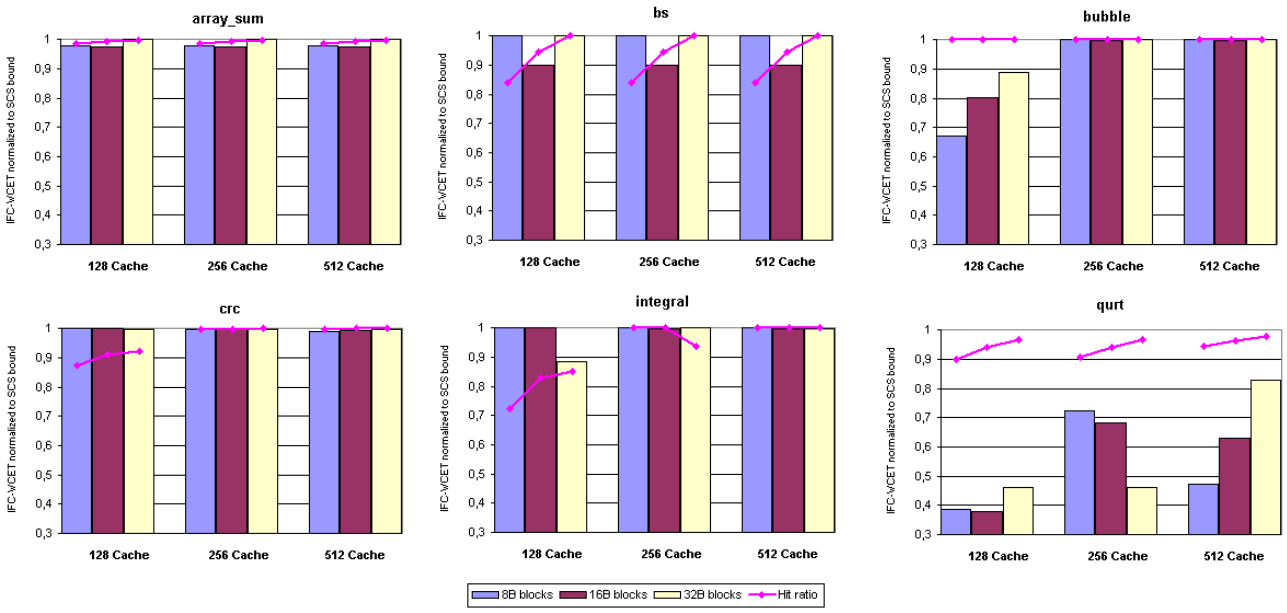

Figure 2. Exact IFC-WCET normalized to the SCS IFC-WCET bound, and instruction hit ratio (line).

the path, cache content may present much more variations. As we understand, this benchmark is more similar in complexity to real codes than the others, so that it is expected that real codes would obtain similar IFC-WCET reductions.

3.2 On the Exact Analysis

Table 1 shows data about the performed analysis. The first columns show the size, cumulative number of iterations in the worst case path and number of possible execution paths, showing the combinatorial explosion problem. Application of Proposition 3 is shown in the next columns, containing the maximum and average number of relevant paths (cache states) considering the previous bound. The *max.* value accounts for the maximum number of relevant paths after any pruning point of the program, and the *avrg.* is the average number of relevant paths from the beginning to the end of the program. Next, for every cache and block size it is shown the maximum and average number of relevant paths in the analysis. Remember that, with our method, only these relevant paths are analyzed. Notice also that these two numbers are below the bound on the number of relevant paths.

It can be seen that the resulting number of analyzed paths is up to more than $2\,800$ orders of magnitude under the possible paths. Note also that this grows when block size shrinks. The reason is that when blocks are small, the blocks traversed by different paths are very specific, whereas with larger blocks, different paths may end up bringing the same big instruction block to the cache.

Regarding *qurt*, it shows a consistent growing in relevant paths as the cache size varies. This reaffirms our claim that exact analysis is more important as the benchmark gets more complex and the cache becomes bigger.

Finally, execution time of the analysis running on a 3.4 GHz Pentium4 is presented (*Anal. Time*). Analysis times are shown with the only goal of demonstrating that an exact analysis is feasible, since the implementation is not yet optimized in any way. The most computation-intensive part is the analysis of loops, as can be seen by considering the number of iterations in *bubble*, *crc*, and *integral*, and realizing that they also have the highest analysis times. A very effective optimization would be to loop just the number of iterations required to reach the maximum number of states and to obtain the subsequent worst path, instead of performing the full number of iterations as our analysis currently does. Such an optimization would reduce very much the analysis execution time.

4 Conclusions

In this paper we give the theoretical foundations to carry out an exact analysis of the instruction fetch contribution to the WCET (IFC-WCET) in presence of LRU instruction caches. With such an analysis it is possible to compute the exact IFC-WCET instead of the overestimated upper bound that conventional methods may compute.

First, we have proved that the inherent exponential complexity of analysis can be drastically reduced. The IFC-WCET analysis of loops containing branches does not de-

Bench.	Size	Iter.	Possible Paths	Bound on Relev. Paths		Block Size	128 B Cache			256 B Cache			512 B Cache		
							Relev. Paths		Anal.	Relev. Paths		Anal.	Relev. Paths		Anal.
				Max	Avrg		Max	Avrg	Time	Max	Avrg	Time	Max	Avrg	Time
arraysum	152B	100	$\sim 10^{30}$	3	2.99	8B	2	2.00	0.03s	2	2.00	0.02s	2	2.00	0.02s
						16B	1	1.00	0.02s	1	1.00	0.03s	1	1.00	0.07s
						32B	1	1.00	0.10s	1	1.00	0.08s	1	1.00	0.03s
bs	112B	4	16	1	1.00	8B	1	1.00	0.02s	1	1.00	0.01s	1	1.00	0.02s
						16B	1	1.00	0.01s	1	1.00	0.01s	1	1.00	0.02s
						32B	1	1.00	0.01s	1	1.00	0.01s	1	1.00	0.02s
bubble	160B	5 050	$\sim 10^{1\,529}$	9	4.10	8B	6	4.00	0.37s	3	2.03	0.19s	3	2.02	0.20s
						16B	5	3.04	0.27s	3	2.02	0.19s	3	2.02	0.20s
						32B	3	3.00	0.27s	3	2.02	0.19s	3	2.02	0.19s
crc	560B	2 082	$\sim 10^{1\,996}$	216	6.87	8B	14	1.08	0.17s	114	1.77	0.26s	126	6.73	0.95s
						16B	6	1.04	0.16s	44	1.13	0.18s	51	4.62	0.64s
						32B	2	1.00	0.16s	4	1.02	0.16s	9	1.52	0.22s
integral	420B	3 000	$\sim 10^{2\,873}$	87	19.36	8B	19	3.01	1.15s	42	5.34	2.00s	34	7.34	2.73s
						16B	10	2.34	0.85s	14	3.01	1.09s	14	4.34	1.58s
						32B	4	1.00	0.38s	7	1.00	0.38s	5	1.00	0.37s
qurt	752B	60	$\sim 10^{44}$	553	65.29	8B	10	2.29	0.03s	63	12.21	0.07s	281	36.16	0.18s
						16B	4	2.14	0.03s	23	2.73	0.04s	124	15.76	0.09s
						32B	4	2.95	0.03s	18	3.38	0.04s	73	9.94	0.06s

Table 1. Size, number of iterations, possible execution paths, bounded maximum and average relevant paths, and data recorded during the analysis: maximum and average number of relevant paths (those actually analyzed, which have different cache states) and analysis time.

pend on the number of iterations, but on the number of branches inside the loop. This basic bound is also extended to consider set-associative caches, nested loops and multiple states entering a loop. Applying these formalisms to the codes we have studied, the number of possible execution paths and the number of paths relevant for the exact IFC-WCET analysis differ up to more than $2\,800$ orders of magnitude.

Second, we have built a proof of concept tool and run it on several benchmarks. Across several instruction block and cache sizes, the exact IFC-WCET has been compared to the bound provided by Static Cache Simulation (SCS), a widely used approach. In simple benchmarks SCS bounds are very accurate. However, for larger and more complex benchmarks such as *qurt* the exact IFC-WCET is significantly lower than the SCS bound across all block and cache size (up to 62% lower).

Finally, it is important to notice that having a set of benchmarks with exact IFC-WCETs would allow the scientific community to have more concrete WCET references, since currently there is no way to demonstrate how good the results provided by a WCET bound analysis method are.

References

[1] Benchmarks maintained by Mälardalen WCET group. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[2] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, December 1994.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[4] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, November 1999.

[5] C. Healy, D. Whalley, and M. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[6] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[7] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, May 2000.

[8] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of 2nd International Workshop on Worst-Case Execution Time Analysis (WCET'02)*, 2002.

[9] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, Nov. 2007.

[10] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2-3):157–179, May 2000.

[11] R. Wilhelm et al. The determination of worst-case execution times-overview of the methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.