Tightening the WCET Bound through Path Pruning

L. C. Aparicio, J. Segarra, C. Rodríguez, V. Viñals, and J. L. Villarroel

I. Introduction

The WCET computation is one of the main challenges in hard real-time systems, since all further analysis is based on this value. In this work we describe a general method aimed to obtain the exact WCET of a task using an integrated timing and program control flow analysis. Our method is based on following all possible paths and discarding, as soon as possible, those not relevant to the final WCET. This study focus on the analysis of an LRU instruction cache. As a result, we provide several experimental data to confirm that such an analysis is feasible. We also discuss how each cache parameter affects the WCETs of several benchmarks.

As far as we know, there is no computationally acceptable method to obtain the exact WCET in presence of caches. Instead, existing approaches try to compute a safe upper bound of the WCET. Depending on the analysis method, the analyzed code, the hardware, etc. this obtained bound may be near the exact WCET or may even coincide with it, but in other cases the estimated bound may be very far from the real WCET. Unfortunately, in general the accuracy of an WCET bound (and thus the precision of a WCET bounding method) is not known, just because the exact WCET cannot be computed. This means that even well known methods that are assumed to provide accurate WCET bounds may be overestimating very much the WCET in real programs, where the accuracy could not be tested.

In the literature we find different methods to approach the WCET problem in presence of caches [16]. 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 [3], [4], [5], [6], [9], [10], [11], [14], [15]. 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 particular 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 [10]). 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 resulting WCET bound.

In this paper we propose a method aimed towards an exact computation of the WCET. The situation on real-time/embedded systems is such that:

1. The goal is to obtain a very accurate WCET bound (or the exact WCET) to maximize the schedulability of the system and even to execute more or longer tasks.
2. On (mass production) devices where custom hardware is affordable, an accurate WCET bound (or the exact WCET) allows to use cheaper microprocessors (reducing costs per device) and decrease energy consumption (e.g. reducing clock frequency).
3. Many real-time programs are designed once and
used many times without changes, so large compilation and analysis times are acceptable\(^3\) considering the previous goals.

Our method is based on exhaustively analyzing all possible execution paths using an integrated timing and flow analysis. At first glance, such analysis has an exponential complexity, since the number of possible execution paths is doubled each time a branch is reached. However, theoretical properties allows us to discard many possible paths, ensuring that they are not relevant for the WCET analysis [2], [7]. Thus, the analysis can be optimized very much by pruning these execution paths by a branch and bound algorithm. Using this method, the analysis to get the WCET does not need to analyze the full number of possible execution paths, but only a bounded subset of them. Currently our analysis is focused on the instruction cache, but it is meant to be general and to account accurately for the execution costs of other hardware structures in the future, such as a pipelined data path or a data cache.

The paper is structured as follows. In Section 2 we present our method and implementation. Results showing the method feasibility and how the different parameters of an instruction cache affects the WCET are shown in Section 3. Finally, our conclusions are presented in Section 4.

II. Our Method

Let us first define the concept of execution history used below. Given a program under analysis, we call an execution history at a given execution point of the program under analysis to the set of elements that summarizes the WCET analysis from the beginning of the program to that execution point through a particular path. An execution history must allow to resume the WCET analysis and follow any valid instruction path afterwards. It is composed of the PC, to indicate the current execution point, the stack of return addresses, to be able to return from subroutines, the hardware state and the cumulative WCET (CWCET). The hardware state models the state of the components to analyze, such as pipeline, cache tags, etc. Finally, the CWCET is the WCET until this point reached by this execution history. Since the PC, stack of return addresses and hardware state are always used together, we call these elements the execution history state. Note also that this is a concrete state and does not have any abstraction, since even data dependent hardware states have a concrete worst case.

Our goal is to obtain the exact WCET, and not a WCET bound. This implies that every possible execution relevant to the computation of the WCET must be analyzed, and then select the one with longer time as the exact WCET of the program. Essentially, we start analyzing every possible execution and, whenever it can be proved, discard those execution histories that will not reach the WCET.

\(^3\)In the benchmarks that we have tested, our suggested analysis usually provides the WCET in less than a second.

To do this we use two integrated activities of analysis. The first one is the timing analysis, which computes the time required for executing instructions of an execution history. The second one is the program control flow analysis, which creates and discards execution histories to feed the timing analysis. This integrated behavior is similar to the Symbolic execution [9], but the flow analysis is essentially different, since we do not deal with symbolic or partially unknown states. Instead, in order to avoid WCET overestimations, our analysis should not make use of path merging, despite in general this implies an exponential complexity. We consider that our proposed method, being general enough to be applied to any hardware component, is already an interesting contribution. However, the main contribution will be to avoid the exponential complexity when supporting particular hardware components. For instance, modelling an in-order pipeline seems very affordable, because it is only required to “remember” the pipeline state of a few instructions, but supporting components such as instruction or data caches seems far more complex. In this paper we focus on the instruction cache.

A. Timing Analysis

The timing analysis is perhaps the simplest part of our method, since it does not introduce any new concept. Our analyzer gets a cumulative WCET, an execution history state and an instruction, and returns the updated CWCET and the updated execution history state after executing the instruction, according to the hardware model. Note that this analysis does not execute the instruction (as Symbolic execution does), but simply accounts the changes its execution produces. Anyway, if needed, real instruction execution would be carried out in the timing analyzer. This could be the case for modelling a data cache in presence of non-data-dependent memory references.

B. Flow Analysis

The flow analysis manages the different possible execution paths of the program. This analyzer is very different from those found in other methods, which make a static analysis separated from other analysis parts. In our case, the flow analysis is continuously interacting with the timing analyzer, and dynamically creates and prunes execution histories to analyze. Essentially, our flow analysis fetches instructions in program order and takes decisions when certain instructions are reached. For instance, when reaching a conditional branch instruction with an unknown outcome, the flow analysis will fork the current execution history into two different execution histories, so that one of them will take the branch and the other will not take it. In this way, the analysis of every possible execution path is guaranteed. Infeasible paths are meant to be avoided through external annotations. Following the control flow of the program allows us, to certain extent, to avoid a high level control-flow analysis, but anyway some high-
level information such as the loop bounds or the possible indirect jump targets is still required.

Additionally to advance and create execution histories, our flow analyzer can also discard (also called prune) them. We can prune a given execution history if and only if we can guarantee that such execution history will not generate the WCET, i.e. the execution history is not relevant to the WCET analysis. Essentially, pruning involves comparing two (or more) execution histories at a common point (pruning point) and prune the ones with lower CW CET whenever possible. This implies that the analysis of the different execution histories must be done in a synchronized way, so that some execution histories “pause” their advancement and wait for other execution histories to meet them to test whether some of them can be pruned. This concurrent analysis of execution histories can be done in a number of ways. We do not dig into implementations of this synchronization due to lack of space.

C. Pruning conditions

Currently we have tested several pruning conditions, and the most efficient one is pruning by equal states. It is based on the idea that different execution histories can go through different paths and reach a given common point with the same execution history state. For instance, focusing on the instruction cache, think of two executions following different paths and at some time traversing a common path, so that the basic blocks of the common path completely renew the cache content. At this time, both execution histories have the same execution history (cache) state, so their analysis will be identical for now on. That is, analyzing just one of them is enough, and the analyzed one must be the one with higher CW CET. In other words, the execution history with lower CW CET can be pruned.

D. Proof of concept implementation

Fig. 1 shows a diagram of how the different modules are integrated. It can be readily seen that the analysis does not follow the classical two-step (control flow and timing) approach as other tools do, but the timing analysis depends on the control flow of the analyzed program.

Decoding of instructions is based on the SimpleScalar simulator frontend for ARM [8]. Additionally, since our analysis is based on executable binaries, elements such as the compiler optimizations, the ordering of functions into memory or the possible linker optimizations are taken into account by construction.

With this framework, our tool can obtain the worst path and its exact WCET for the analyzed hardware model. Currently we have implemented functions to account the access to an LRU instruction cache of arbitrary size and associativity and also to account a constant time for execution of instructions and a constant time for data accesses.

III. Results

For testing our tool we have used several benchmarks (Table I) exhibiting conditional sentences inside loops, which is our main analysis difficulty. These benchmarks are a subset of those found in [1] plus array_sum and integral, and all they are commonplace in WCET studies [5], [9], [10], [11], [14]. Benchmarks too simple or too similar to those selected have not been considered. Source codes have been compiled with GCC 2.95.2 -O2 for an ARM7 processor (4B/instruction).

The tool models a very simple processor with 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, as in [13]. 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.

Table I shows the code size, the number of executed instructions and the number of iterations through the worst path for each benchmark, using a 256B 2-way set associative LRU instruction cache with 16B blocks. It also shows how the theoretical number of maintained execution histories (relevant paths) is much lower than the possible execution paths, and also that the actual number of relevant paths when applying the benchmarks on a particular cache is lower than its theoretical value. The last column shows the required analysis time. Results

2Simple benchmarks without if-then-else statements in loops include, for example, FIR, JPEG forward DCT, FFT, matrix sum/multiplication and Fibonacci series. Benchmarks with just if-then statements generally do not represent conditional sentences for the WCET analysis, since taking the branch is constant time for data accesses.

3Experiments performed assuming other miss penalties and data cache behavior, such as hit-always, may change the worst case path, but the trends observed in the WCET remain.
show that the exhaustive integrated analysis is perfectly affordable for the benchmarks and the hardware considered. This may indicate that the goal of obtaining the exact WCET when adding the analysis of other complex hardware components may also be feasible.

We also include the WCET results for several benchmarks in a more detailed way in Fig. 2. Each one of these figures shows the WCET in function of the cache parameters. Cache size corresponds to the main x-axis, dividing the figure into separated regions corresponding to each particular cache size. Each one of these regions shows three plots representing the three block sizes (8B, 16B and 32B), and each one of these plots shows the WCET depending on the cache associativity, increasing the associativity degree from left to right, i.e. from direct mapped to fully associative.

These figures show the trends of the WCET respect several instruction cache parameters. As it can be seen, the main tendency is that WCET decreases when the block size grows. This is a very known fact for the cache designer wanting to minimize the miss ratio, see for instance [12]. However, assuming this design rule to the WCET minimization would be incorrect unless exact WCET are managed, since overestimations may seriously disturb any correlation. To the best of our knowledge, these are the first data about instruction cache design not related to WCET bounds.

Regarding the cache size, obviously increasing the cache size also improves the WCET. The flat line appearing when the cache size grows simply indicates that the whole benchmark fits in cache, so adding more size does not improve results.

Trends on associativity are not so uniform. In general, as the associativity increases the WCET becomes worse. Again, this result is not new in the instruction cache context. It is well known that a perfect loop slightly greater than the instruction cache behaves very different in a direct mapped cache than in a fully associative cache with LRU replacement. In the first case the hit ratio approaches to 100% whereas in the second one tends to zero. However, some combinations of cache and block sizes have the opposite trend (qurt-128B is an extreme example). This opposite results can only be explained by assuming that two chunks of code are been executed alternatively and are mapped on the same sets, so that they evict each other in the direct mapped cache, and both remain in cache in the 2-way set associative cache.

Finally, notice that all results plotted in Fig. 2 come from the interaction between the instruction cache and the worst execution path, and that such concrete path may depart notably from the “average” behavior reported in the cache design literature. Therefore, it is important to design the instruction cache from an exact WCET analysis, not from the standpoint given by the performance-oriented design studies.

Fig. 2. WCET of several benchmarks varying the instruction cache configuration.
A. Limitations

Since all the benchmarks we have tested have been analyzed in a very short time, we have designed several synthetic ones to stress our tool and test its limitations. These benchmarks consist of a loop enclosing several alternative paths. The length of these paths is balanced and long enough to occupy several cache blocks and to avoid the compiler to use predication. The tested instruction cache is selected depending on the program to maximize the number of possible execution history states. Theory proves that, for instruction caches, the number of execution history states in loops containing $p$ alternative paths has a combinatorial explosion, but it is bounded to $\sum_{i=1}^{p} p^i = \sum_{i=1}^{p} p^i / (p-1)$ [2], [7]. Using these benchmarks and pruning by equal states, we have tested that the maximum acceptable value for $p$ in our tool is around 10, reaching a number of non-pruned states in the order of millions.

Although the analysis of non-synthetic benchmarks has been perfectly feasible up to date, we have also considered how to solve an eventual explosion of execution history states. It could be easily avoided by substituting our path pruning by path merging on certain moments. Obviously this solution would introduce overestimation, but would also reduce the number of execution history states.

IV. Conclusions and Future Work

In this paper we present a method to obtain the WCET (and worst path) of a task, using an integrated timing and flow analysis. Our method is based on following all possible paths and discarding, as soon as possible, those not relevant to the final WCET. This relevance is tested through formal properties, so that no path merging is used and thus there is no overestimation. Our results show that the exhaustive integrated analysis is perfectly reasonable when analyzing an LRU instruction cache. This may indicate that the goal of obtaining the exact WCET when adding the analysis of other complex hardware components may also be feasible. We also show how the exact WCET is affected when the main parameters of the instruction cache are varied: both increasing the block size and cache size reduces the WCET, but associativity is not desirable for LRU instruction caches in the general case.

A very accurate WCET bound, or an exact WCET, is very important to reduce costs of embedded and real-time systems. Having a set of benchmarks with exact WCETs would allow the scientific community to have concrete WCET references, since currently there is no way to demonstrate how good the results provided by a WCET bound analysis method are.

Our future work is to add support for other hardware components to our tool, such as pipeline, branch predictors and data caches, so that our hardware model becomes more realistic.

Referencias


5th International Workshop on Worst-Case Execution Time Analysis (WCET'05), pages 15–18, 2005.

