Instruction Fetch Contribution to Exact WCET in Systems with Cache∗

L. C. Aparicio†, J. Segarra†‡, J. L. Villarroel†, V. Viñals†‡, C. Rodríguez§

† Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza
‡ Facultad de Informática, Universidad del País Vasco
§ HIPEAC NoE (High-Performance Embedded Architectures and Compilers, FP6-IST-004408)
{luisapa, jsegarra, jvillia, victor}@unizar.es, acprolac@ehu.es

Abstract

WCET computation is one of the main challenges in the study of hard real-time systems, since all further analysis is based on this value. At the same time, modern processors have hardware components with a variable latency not known at compilation time. Caches have this kind of behavior and the difference between a hit and a miss is usually very significant, so accuracy is specially desirable in this case.

In this work we compute the exact WCET in presence of an instruction cache. In the benchmark codes we have studied, the instruction fetch contribution to the WCET is up to 62% lower than its computed bound, and the difference between the possible execution paths and the ones relevant for the exact WCET analysis is extremely large.

1 Introduction

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

∗This work has been supported by the project "Grupo Consolidado de Investigación" of the Diputación General de Aragón and by the projects TIN2004-07738-C02-02 (50% ERDF/FEDER) and DIP2003-07986 of the MIC/MCyT of Spain.

Given hardware components with a fixed latency, the WCET can be computed from the partial WCET of each basic block of a program. For example, the WCET of a loop enclosing several alternative paths can be computed as the product of the number of iterations by the execution time of the longest path within an iteration. However, in order to improve performance, modern processors have hardware components with a variable latency which depends on the previous operations performed by such components (e.g., caches, branch predictors, pipelined datapaths, etc.). This kind of hardware is based on the well known idea that execution is very repetitive, and on caches this means execution time is improved by taking profit of the reuse of instructions and data. However, this expected reuse is not guaranteed, and it is needed to analyze each and every execution path to verify it and thus compute the exact WCET.

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 WCET. Nevertheless, some of these resulting estimates provide, in some cases, the exact WCET [8].

In the benchmark codes we have studied, the instruction fetch contribution to the (exact) WCET is up to 62% lower than its compared bound, and the difference between the possible execution paths and the ones relevant for the exact WCET analysis is up to more than 1500 orders of magnitude.
This paper is structured as follows. In Section 2 the WCET problem is outlined and existing approaches are sketched. We describe our analysis method in Section 3. In Section 4 we show our obtained results and compare them with the well-known Static Cache Simulation method in presence of an instruction cache [9]. Finally, in Section 5 conclusions and future work are presented.

2 Motivation and Related Work

WCET of a task is required for schedulability analysis in real-time systems and, more recently, for real-time virtual processors [1, 5]. Essentially, the WCET can be computed by accounting the time requirements (processor cycles) for every possible execution path in the task and then selecting the one requiring more time. However, this is very expensive due to the exponential complexity of the problem. A conditional sentence with two paths inside a loop with just 100 iterations has \(2^{100}\) different execution paths.

Currently, modern processors use cache memories to bridge the increasing gap between the ever-faster processor and the moderately faster memory. Cache memories are small and very fast buffers of instructions and data. They are used for decreasing the average access time and reducing the power consumption. However, the behavior of a cache is not easily predictable in compilation time since its content depends on the previously taken path. This means that to know the cache content at a given execution point we need to know, for every execution path reaching this point and for every memory reference in these paths, whether it is a hit or a miss. Obviously, the exponential number of possible paths makes the problem too complex both in time and space.

Figure 1 shows an example on the importance of an exact WCET and the difficulty to compute it. Costs of executing each basic block are based on costs in Table 1 and are shown inside each block. Take into account that possibilities shown in this table are a subset of the actual interdependencies that may occur in real codes. The first row of this table gives the times required to fill caches for the first execution of the paths A and B respectively. Second row accounts for the cost of executing either A after B or B after A, respectively (assume some blocks required in A have been evicted by B or viceversa). The last row accounts for two consecutive executions of A or B respectively. Figure 1(b) shows a naive analysis that does not take into account the path followed in the previous iteration, while Figure 1(c) shows the exact WCET computation by considering all possible paths. Even in such a simple example, notice that the exact WCET is 42 time units lower than the naive WCET bound computation. In general, the only way to reach this accurate result is to analyze every possible path.

In the literature we find different methods to approach the WCET problem in presence of caches [14]. 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, 6, 7, 9, 8, 10, 12, 13]. 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 fused, 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 [9] means a miss followed by successive hits). Since WCET bounds must be upper bounds, any uncertainty must be considered as the worst possible case. At the same time or in a second step, the timing of executing each instruction

<table>
<thead>
<tr>
<th>Execution Case</th>
<th>Path A</th>
<th>Path B</th>
</tr>
</thead>
<tbody>
<tr>
<td>First execution</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>Alternated ex.</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>Two consecutive ex.</td>
<td>10</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1: Execution costs.
for (i = 0; i < 4; i++)
{
    if (cond[i])
    {
        Path A
    }
    else
    {
        Path B
    }
}

(a) Loop with a conditional.

WCET
Path B
Path A

(b) Assuming first execution always.

WCET + 4 \times 40

(c) Exact WCET computation.

Figure 1: Computation of the WCET (4 iterations) with caches.

is considered to compute the resulting WCET bound.

Digging into concrete techniques, Cycle-level Symbolic Execution performs a cycle-level analysis of control flow, so that it can deal accurately with timing related to architectural components such as pipelines, functional units or caches [8]. This is done using symbolic data, since actual values are not known at compilation time. This symbolic execution allows a high-level path analysis, which is used to discard infeasible paths. The main drawback of this method is that, in order to avoid the exponential growth of the number of paths, whenever several paths reach the same position in the program and none of them can be guaranteed to be worse than the others, paths are merged. At this time, their analysis loses information on the previously taken path (and thus some content in cache), which may lead to pessimistic estimates. Abstract Interpretation uses the semantic properties of programs, thus supporting correctness proofs of program analysis [4]. This technique can be used to estimate cache behavior, being able to classify each access as hit, miss or unknown [6, 12]. Path merging is present in this technique by using a list of contents that must or may be in cache, losing information on the previously taken path as above. Static Cache Simulation provides a more detailed classification on each memory reference [3, 7, 9]. This classification combined with a control-flow graph allows to compute a WCET bound. To reduce the combinatorial explosion of cache states Static Cache Simulation defines an Abstract Cache State, which is used to obtain the worst cache state at a given execution point. This abstract cache state is pessimistic by definition, because it merges all paths reaching this point, also losing the information on the previously taken path. We compare our results with this method in Section 4.

As outlined above, none of these WCET computation methods is targeted at obtaining an exact WCET. Instead, they assume that an exhaustive analysis is too complex, and thus they focus on obtaining an upper bound as close as possible to the real WCET. However, not all possible paths are relevant to compute the WCET, but only those which lead to different states [2]. This is specially important in loops, where the number of different paths can grow exponentially with each iteration.

3 Our Method

In order to make exhaustive analysis we have used a symbolic simulation [14] tool, which follows (without actually executing) the control flow in the analyzed benchmark. Following the control flow is simply decoding instructions sequentially and following unconditional branches. For conditional branches, annotations (e.g. number of iterations in a loop) are 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 pruning all paths which would never be the worst one [2]. Essentially, this means that every path analysis must stop at a pre-defined pruning point. When every single path analysis reaches this point, cache states are compared. If two path analysis have an identical state, the path with a lower cumulative WCET until that point can be pruned. This is specially effective inside loops, where instruction cache states are bounded by \( \sum_{i=1}^{\min(p,n)} \frac{p!}{(p-i)!} \), where \( n \) is the number of different paths inside the loop. This means that the number of different states at the end of each iteration shall always be pruned to this bound. This optimization can be repeated by adding more pruning points, but effectiveness is dependent on their placement.

Notice that our tool does not make any analysis on infeasible paths, but relies on external annotated data. This means that if annotations do not avoid infeasible paths, our tool will analyze these paths and the obtained WCET may be overestimated.

4 Results

In this section we describe the obtained results applying the previous analysis optimizations.

We have tested a subset of benchmarks that have been used in other works [6, 8, 9, 10, 12]. We have selected a subset of those which have conditional sentences, since any benchmark without them\(^1\) would not need a tool for obtaining its WCET in presence of instruction cache. That is, the exact WCET in presence of instruction cache of any program without conditional sentences can be obtained by a mathematical expression. Table 2 shows the list of the tested benchmarks, with their static instruction size and the number of executed instructions through the worst path. Source codes have been compiled with GCC2.95.2 -O2 for ARM.

We have modeled a simple processor which processes instructions in two sequential stages: instruction fetch and execution. Instruction fetch takes 1 or 60 cycles depending on whether it hits or misses in the instruction cache [11]. 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.\(^2\)

Our cache configurations have sizes of 128, 256 and 512 bytes, with block sizes of 8, 16 and 32 bytes. All of them are 2-associative caches.

4.1 Comparison between Exact WCET and SCS WCET 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, for this WCET value we subtract the time contributions of data accesses and execution phases. That is, we take just the Instruction Fetch Contribution to the WCET (IFC-WCET). Figure 2 shows results by cache size and block size. Bars represent the IFC-WCET obtained with our method, normalized to the IFC-WCET bound obtained by the Static Cache Simulation (SCS) method [9].

\(^1\)Benchmarks without conditional sentences include, for example JPEG integer implementation of the forward DCT, matrix sum, matrix multiplication (integer and float) and Fibonacci series function.

\(^2\)Experiments performed assuming other data cache interactions, such as always hitting, may change the worst case path, but the trends observed in the instruction fetch contribution to WCET remain.

<table>
<thead>
<tr>
<th>Bench</th>
<th>Description</th>
<th>Size</th>
<th>Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_sum</td>
<td>Sum of array elem.</td>
<td>162 B</td>
<td>1747</td>
</tr>
<tr>
<td>bs</td>
<td>Binary search</td>
<td>112 B</td>
<td>56</td>
</tr>
<tr>
<td>bubble</td>
<td>Bubble sort alg.</td>
<td>160 B</td>
<td>95.178</td>
</tr>
<tr>
<td>crc</td>
<td>CRC</td>
<td>560 B</td>
<td>45.711</td>
</tr>
<tr>
<td>integral</td>
<td>Comp. 3 integrals</td>
<td>420 B</td>
<td>141.102</td>
</tr>
<tr>
<td>sqrt</td>
<td>Roots of quad. eq.</td>
<td>752 B</td>
<td>1715</td>
</tr>
</tbody>
</table>

Table 2: Set of benchmarks with their static instruction size and executed instructions through the worst path.
Figure 2: Instruction Fetch Contribution to exact WCETs normalized to SCS WCET bound and the instruction hit ratio (line) for several 2-associative cache configurations.
ures also show the instruction hit ratio. That is, the ratio of instruction hits out of the total instruction fetches through the worst path.

Reductions in IFC-WCET are produced when WCET bounding methods (SCS in this case) are not able to accurately classify whether an instruction fetch is a hit or a miss. In these cases, any bounding method provides an upper/safe bound which increases their WCET bound, whereas our exact WCET computation is able to provide the exact value.

Most benchmarks show a minimal reduction not always appreciable in figures (less than 0.03). In these cases SCS results are very accurate.

However, on benchmarks that contain many branches (especially if they are inside loops), bounding methods are forced to merge different paths into a single one, and this new one must be as bad as (or worse than) the worst of the previous ones. This means they lose information (e.g., detailed cache content) which later may imply more uncertainty and thus more upper bounds.

Regarding the qurt benchmark, note that in this case reduction is much more significant (up to 62%). This benchmark is much bigger than the others, which means that it does not even fit in our biggest cache. Thus, depending on the taken path, cache content may present much more variations. As we understand, this benchmark may be more similar to real codes than the others, so that it is expected that real codes would obtain similar WCET reductions. It is in these cases where one can take profit of an exact WCET computation.

Notice that the quality of the WCET bound depends on both the block and the cache size. Indeed, seeing the qurt figure, although the instruction hit ratio growing is consistent with the block size, the deviation between the exact IFC-WCET and the reported bound does not follow any trend.

The hit ratio also present in figures is specially interesting when reductions are minimal. Notice that, although these benchmarks (and many times much simpler ones) are commonly used in WCET papers, they are very simple, even using cache sizes much smaller than those found in real systems. In most of cases, when our IFC-WCET reduction is minimal this hit ratio is very high. This means that the benchmark code is guaranteed to be in cache most of times. In these cases, an accurate timing can be easily computed. On the other hand, lack of precision for the few times these hits are not guaranteed has very little impact on the global IFC-WCET.

4.2 Execution Time of the Analysis

Table 3 shows several data about the performed analysis. In the first place, it shows the number of different execution paths of each benchmark. These paths show the combinatorial explosion problem. Next, for every cache size, it shows the number of paths actually relevant for the WCET (those we analyze) and the execution time of our analysis. Both these values are the maximum of the three block sizes analyzed. Analysis have been performed on a 3.4GHz Pentium 4.

These values show that our proposal to compute the exact WCET is highly bounded and, in our benchmark set, the number of paths whose analysis can be saved has up to more than 1500 orders of magnitude. Moreover, take into account that our implementation is not optimized. In fact, the reported times include storing long records in disk for debugging purposes. That is, time values in this table are shown with the only goal of demonstrating that our analysis is perfectly feasible, but not to provide an execution time reference.

5 Conclusions and future work

In this paper we propose a new approach for the WCET analysis of a program. Our approach is targeted at obtaining the real WCET instead of a WCET bound. This goal is based on making a detailed integrated analysis and avoiding the exponential complexity by theoretical properties, so that no useful information is lost.

We have studied several benchmarks, and provided results for them. They show that currently many benchmarks are too simple,
and in these cases WCET bounds may be very accurate. However, for larger and more complex benchmarks such as the computation of roots of quadratic equations (qurt) the real WCET is much lower than its computed bound using other methods. In the benchmark codes we have studied, exact WCET is up to 62% lower than its compared bound, and the difference between the possible execution paths and the ones relevant for the exact WCET analysis is up to more than 1500 orders of magnitude.

Regarding future work, the analysis of data caches is also required. Our method is currently valid for static data, but performance on variable references (i.e. arrays, pointers, etc.) is yet to be addressed. A detailed study on tuning the application of our method is also needed, that is, to analyze the most convenient locations to place our pruning points.

Finally, it is important to notice that 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.

References


<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Possible Paths</th>
<th>128 B Cache</th>
<th>256 B Cache</th>
<th>512 B Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_sum</td>
<td>1.27×10^{20}</td>
<td>2</td>
<td>2s</td>
<td>1</td>
</tr>
<tr>
<td>bs</td>
<td>1.6×10^{1520}</td>
<td>5</td>
<td>4m27s</td>
<td>3</td>
</tr>
<tr>
<td>bubble</td>
<td>1.6×10^{932}</td>
<td>14</td>
<td>1m38s</td>
<td>114</td>
</tr>
<tr>
<td>crc</td>
<td>8.5×10^{2572}</td>
<td>10</td>
<td>21m23s</td>
<td>42</td>
</tr>
<tr>
<td>integrales</td>
<td>3.4×10^{44}</td>
<td>10</td>
<td>6s</td>
<td>63</td>
</tr>
</tbody>
</table>

Table 3: Possible execution paths and analysis data for the tested benchmarks. Data include the maximum number of relevant paths (may be lower depending on the cache block size) and analysis times (may be lower depending on the cache block size) for different cache sizes.


