Execution Path Pruning for WCET Analysis

L. C. Aparicio, J. Segarra[1 2], J. L. Villarroel[1], V. Viñals[1 2]
Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza.
[1]I3A (Instituto de Investigación en Ingeniería de Aragón)
[2]HiPEAC NoE (High-Performance Embedded Architectures and Compilers)
{luisapa, jsegarra, jlvilla, victor}@unizar.es

Abstract— **WCET computation is one of the main challenges in the study of HRTS, since it is needed to guarantee the time requirements. Moreover, modern processors have hardware components with a variable latency not known at compilation time which makes the problem even harder. In particular, the WCET computation problem in presence of caches takes exponential complexity.**

In this work we propose two techniques targeted to compute WCET accurately in presence of both instruction and data caches. Both techniques reduce drastically the number of states to analyze by pruning all the paths located outside the time-critical path.

Keywords— **Hard Real Time Systems, Cache memories, Worst Case Execution Time.**

I. Introduction

Worst Case Execution Time (WCET) computation is one of the main challenges in the study of Hard Real Time Systems (HRTS). The WCET is difficult to determinate since it depends on both the hardware and the software, but it is needed to guarantee the time requirements of HRTS.

For 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.

On the other hand, to improve performance modern processors have hardware components with a variable latency dependent on the past, e.g. caches, branch predictors, pipelined execution, etc. In these cases it is needed to analyze each and every execution path to compute the WCET, storing and updating throughout all that paths the relevant hardware state as dictated by the interaction between program execution and hardware behavior.

As far as we know, it does not exist any approach to obtain the exact WCET in presence of caches. Instead, existing approaches try to compute a safe upper bound of WCET, either by means of analysis [2], [4], [6], [8], [11], [12], [16], [17], [21], [22] or by limiting the cache dynamics by locking their contents [9], [10], [13], [14], [15], [18], [19].

The contribution of this work consists of two techniques aimed towards an exact computation of WCET in presence of both instruction and data caches. Our techniques can be applied in the convergence points of the program control flow, for instance, in the first instruction reached by different execution paths. The ending of an if-then-else construct is typically one of these convergence points. The first technique (*equal-cache path removal*) allows to discard all those execution paths with equal cache state, keeping only the one with the longest accumulated execution time. Our second technique (*arbitrary-cache path removal*) computes the difference of the accumulated execution time between two execution paths. If this difference is bigger than a threshold (dependent on the cache state) we can safely remove the execution path with smaller accumulated execution time.

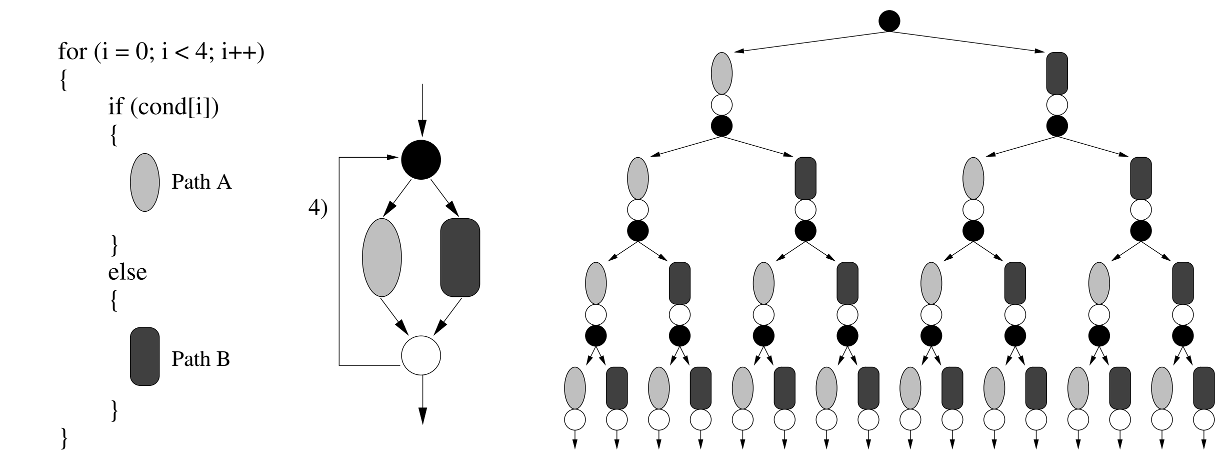
This paper is structured as follows. In Section 2 our work is motivated and the existing approaches are sketched. In Section 3 we describe our first technique: *equal-cache path removal* and we analyze the problem complexity on loops. Our second technique (*arbitrary-cache path removal*) is described in Section 4. Finally, in Section 5 conclusions and future work are presented.

II. Motivation and related work

Obtaining WCET is required for a schedulability analysis in HRTS. A safe (upper) bound of the WCET of a task can be calculated by static analysis. This can be accomplished by accounting the time requirements (processor cycles) for every unique execution path in the task and then selecting the longer one. This is very expensive due to the exponential complexity of the problem. See for example how a piece of code with a conditional sentence inside a loop (Fig. 1 (a)) represents a control-flow graph (Fig. 1 (b)) which shows a combinatorial explosion of paths when unrolled (Fig. 1 (c)). 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 ever-faster processor and 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 contents depends on the path taken during program execution. It is difficult to statically compute which blocks are inside the cache at a given instant, since it is equivalent to compute for every memory reference and for every execution path if it is either a hit, an com-

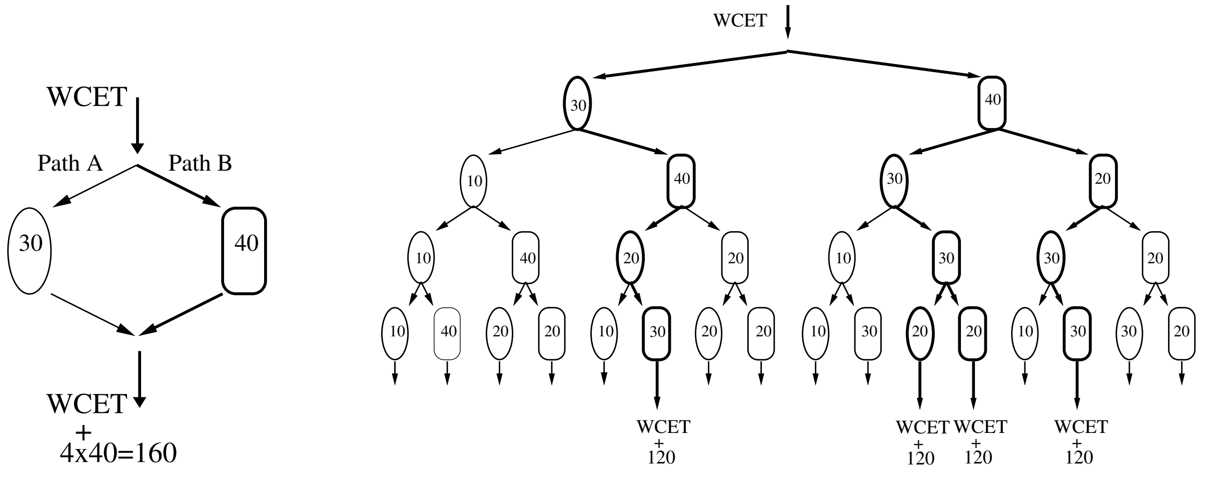
This work has been supported by the project “Grupo Consolidado de Investigación” of the Diputación General de Aragón and by the projects TIN2004-07739-C02-02 (50% ERDF/FEDER) and DIP2003-07986 of the MEC/MCyT of Spain.



```
for (i = 0; i < 4; i++)
{
    if (cond[i])
    {
        Path A
    }
    else
    {
        Path B
    }
}
```

(a) Loop with a conditional (b) Control−flow graph (c) Combinatorial explosion of paths after 4 iterations

Fig. 1. Combinatorial path explosion in a loop enclosing two alternative paths



(a) Assuming first execution always (b) Exact WCET computation

Fig. 2. WCET computation with caches

Execution Case	Path A	Path B
First execution	30	40
Alternated execution	20	30
Two consecutive executions	10	20

Table I: Execution Costs

pulsory miss, a capacity miss or a conflict miss [7]. Obviously, the exponential number of required cache states makes the problem too complex both in time and space.

Fig. 2 shows the code in Fig. 1 executing with the costs given in Table I. The first row of Table I gives the times requited 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 (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. Fig. 2 (a) shows a naive analysis that does not take into account the path followed in the previous iteration, while Fig. 2 (b) shows the exact WCET computation by considering all possible paths.

In the literature we find different methods to approach the WCET problem in presence of caches. Due to the difficulty of the problem, most of these methods try to solve it by dividing the problem in two seemingly simpler steps. In the first step, they avoid the combinatorial explosion of cache states by not always remembering the whole history of the followed path. This allows to classify each memory access either as a hit or a miss. However, this classification is pessimistic because there has been a loss of information associated to the reduction of cache states. In the second step, they compute an upper bound of the WCET analyzing all execution paths and considering the worst case in each memory access (obtained in the first step). This means the WCET of a task can be widely overestimated in general.

Next, we summarize several approaches that can be used to analyze the WCET in presence of caches.

A. Analytical methods

These methods make a static analysis to classify all memory accesses in the worst case. Later they perform a timing analysis to calculate the WCET. *Cycle-level Symbolic Execution* performs a cycle-level analysis of control flow, so that it can deal with timing related to architectural components such as pipelines, functional units or caches. This is done using symbolic data, since actual values are not known at compilation time. However, their cache analysis is quite pessimistic [8]. *Abstract Interpretation* uses the semantic properties of programs, thus supporting correctness proofs of program analysis [3]. This technique can be used to estimate cache behavior, being able to classify instruction executions as *hit*, *miss* or *unknown* [4], [17]. *Static Cache Simulation* provides a more detailed classification on each memory reference [2], [6], [12]. These classifications combined with a control-flow graph allow to compute a WCET bound. To reduce the combinatorial explosion of cache states *Static Cache Simulation* defines an *Abstract Cache State*, which represents the worst cache state at a given execution point. This abstract cache state is pessimistic by definition, so the WCET obtained with this method is overestimated. Additionally, this method has particular problems on data caches, since it needs regular data accesses. This is partially solved looking for data access patterns either directly [21], [22] or using *Cache Miss Equations* (CME) [5], [16].

B. Restrictive methods

Restrictive methods assume the existence of commands or instructions whose execution locks the cache content until another command resumes regular replacement (unlock). As long as the cache is in a locked state, hits are serviced normally, but no new block is allowed to enter into the cache. This allows a precise access classification (hit/miss) because the cache content is known and fixed until the unlock instruction resumes normal behavior.

Several authors propose to lock the cache during the whole system execution [13], [15]. In this case low complexity algorithms are needed to select the content to fix [14]. Genetic algorithms have been also applied on this subject [10]. Other authors use cache locking just on chunks of code where its exact content (and thus its access classifications) cannot be guaranteed [18], [19]. CMEs are used to determine these periods [5], [20].

In general, these methods allow to determine exactly whether each memory access is a hit or a miss. However, this precision is obtained by locking the cache behavior and not taking advantage of it. This means they increase predictability by reducing performance, which can lead to increase the WCET. Additionally, the cost of lock/unlock and cache preloading instructions must also be considered.

III. Path removal with equal cache states

In this section we propose a method in order to discard execution paths reaching identical cache states. This method does not lose any information, and the discarded execution paths are guaranteed not to be the worst case execution paths. Essentially, we analyze the cache state and the accumulated WCET on a given execution point for all execution paths. If some of them have the same cache state, we maintain the one with the longest WCET, discarding the rest.

We focus the application of this method on loops. The number of possible execution paths in a loop with a conditional inside is p^n, where p represents the number of paths in the conditional and n the number of loop iterations. However, not all execution paths lead to different cache states. Therefore, to calculate the WCET of a piece of code containing a loop with several paths inside, we only have to consider the worst execution path for each different cache state at the loop ending.

This method is valid for any cache architecture. In order to be general, in our theoretical reasoning we assume the worst one: a totally associative cache (no conflict misses) with infinite size (no capacity misses). We also use the most common replacement policy: least recently used (LRU), where cache content is ordered by the time of the last reference.

Next we are going to define some key cocepts in order to bound the maximun number of different cache states

Definition 1: (Cache State) Let A be a sequence of memory accesses associated to an execution path of a program. $CS(A)$ represents the cache state that is reached after executing the sequence of accesses A.

For notation purposes, we donete CS(A,B) the CS that is reached after executing first the sequence of access A and later the sequence of access B

For example, if we assume all memory references in the paths are static (do not vary durring the execution) and we consider the execution of all possible execution paths of a loop, with 4 iterations, enclosing 2 alternative paths (A, B) as in Figure 1 (a), the number of different cache states that can be reached is 4 (as is proved in [1]) because:

$CS(A, A, A, A) = CS(A)$
$CS(B, B, B, B) = CS(B)$
{ CS(A,A,A,B), CS(A,A,B,B), CS(A,B,B,B), CS(A,B,A,B), CS(B,A,A,B), CS(B,A,B,B), CS(B,B,A,B) } = CS(A,B)
{ CS(B,B,B,A), CS(B,B,A,A), CS(B,A,A,A), CS(B,A,B,A), CS(A,B,B,A), CS(A,B,A,A), CS(A,A,B,A) } = CS(B,A)

Fig. 3 shows the evolution of all possible cache states reachable after four iterations in the loop in Fig. 1. It also shows that some cache states are duplicated and thus can be discarded. In fact, after the loop output only four (out of 16 possible) different states can appear.

Proposition 1: Let L be a loop containing p different paths inside. Let us assume all memory references in the paths are static (do not vary during the execution). The maximum number of cache states at the ending of L is $\sum_{i=1}^{p} \frac{p!}{(p-i)!}$, that is $\sum_{i=1}^{n} V_p^i$ where V_p^i are the variations of i elements selected from p elements without repetitions.

Proof: This proof is detailed in [1]. ∎

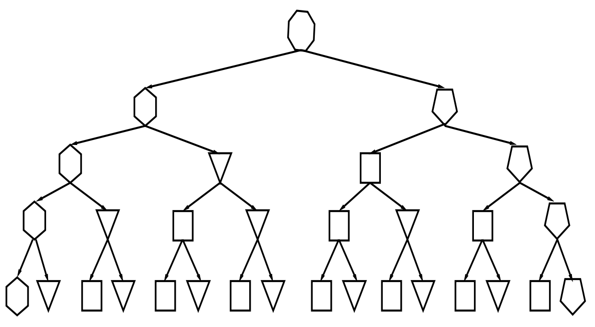


Fig. 3. Combinatorial explosion of cache states in a loop enclosing 2 alternative paths

Fig. 4 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, 8, and 16 alternative paths. Note that although the number of different cache states grows initially exponentially then it is constant, besides it is much smaller than the number of execution paths.

Corollary 1: The maximum number of different cache states that can be reached at the end of a loop does not depend on the number of iterations of the loop (see Fig. 4).

Corollary 2: The maximum number of different cache states that can be reached at the end of a loop depends on the number of paths inside the loop (see Fig. 4)

Definition 2: (Execution Path Analysis State) Let P be a execution path. We define the $EPAS$: Execution Path Analysis State as the couple formed by: the cache state (data an instruction) and the execution time accumulated by the path in a common execution point (a point shared by different execution paths).

Corollary 3: To calculate the WCET considering the EPAS graph we must only take, for each different cache state, the path with the largest accumulated execution time. That is to say, for all paths reaching a common instruction (in this case, the loop exit) with the same cache state, only the one with the largest accumulated execution time can actually be the one with the "real" WCET until that point.

In Fig. 5 we can see the reduction of execution paths obtained from Figure 1 when the EPAS graph is considered. Although the problem complexity remains exponential, it now depends on the number of paths inside the loop, which in general is much smaller than the number of loop iterations.

IV. Path removal with arbitrary cache states

By Corollary 3, if we have several execution paths reaching the same cache state on the same instruction, only one of them (the one with higher WCET until then) will be relevant for the program WCET. This means that all paths reaching different cache states must be analyzed. In this section we propose how to detect (and discard) execution paths with arbitrary cache states which will be irrelevant to obtain the program WCET. We first define the threshold for

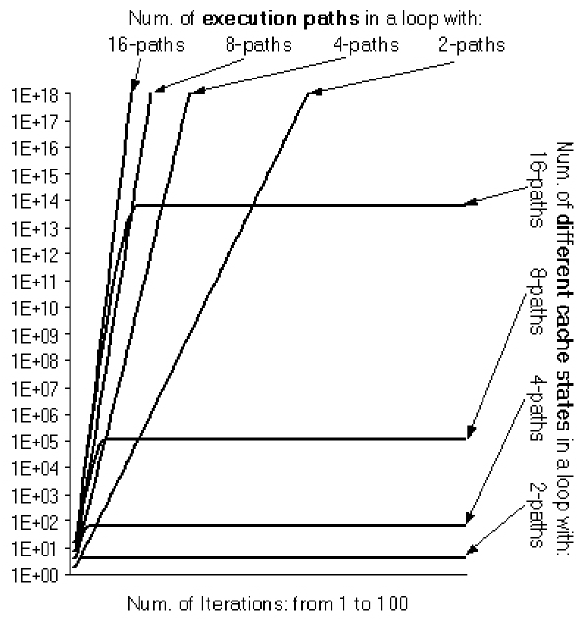


Fig. 4. Possible executions paths and possible cache states according to the number of iterations

discarding paths and then prove these paths can be safely discarded. Next, we propose two methods for obtaining this threshold and finally we discuss the computational cost for both methods.

Definition 3: (Largest Cost Difference on Execution Times) Let us have two different EPAS A, B on the same instruction (e.g. coming from two different paths), and other EPAS C (reachable by both A and B with an unknown WCET) some instructions in the future. We define the $LCDET$: Largest Cost Difference on Execution Times from A to B ($LCDET_{B-A}$) as the *maximum* cost for B to become C ($LCDET_{B-C}$) minus the *minimum* cost for A to become C ($BCET_{A\to C}$) after any incoming instruction sequence on both states A and B (the same sequence on both states), i.e. $LCDET_{B-A} = WCET_{B\to C} - BCET_{A\to C}$.

Proposition 2: Let us have two different paths P_A, P_B which lead to a common instruction with two different EPAS A, B from a common initial situation I. Let us suppose each path accumulates a different worst case execution time $WCET_{I\to A}$, $WCET_{I\to B}$ at that point, and $WCET_{I\to A} > WCET_{I\to B}$. In order to obtain the WCET, path P_B can be safely discarded if $WCET_{I\to A} \geq WCET_{I\to B} + LCDET_{B-A}$

Proof: This proof is detailed in [1]

■

Proposition 3: Let s be the number of sets in a cache, n the number of ways, m the miss cost and h the hit cost ($m > h$). An upper bound of the $LCDET$ for two states of a cache using LRU is $s \times n \times (m-h)$.

Proof: This proof is detailed in [1]

■

Note that this LCDET bound is in fact the cost of refilling the whole cache with misses minus refilling it with hits, which is the largest memory access difference. For example, if we consider the EPAS-

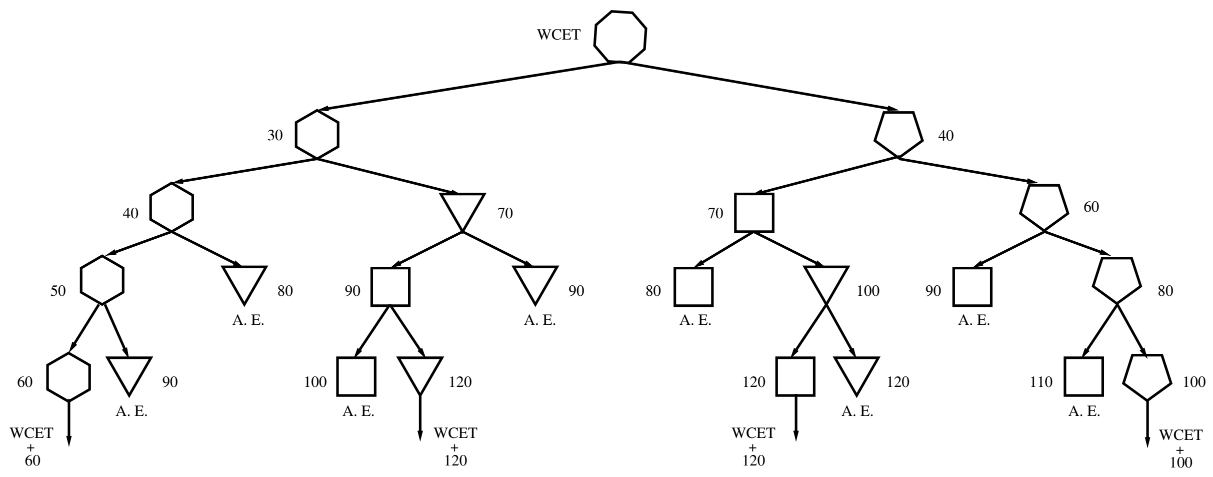


Fig. 5. Path removal with repeated cache states in a loop enclosing 2 alternative paths. A.E. stands for Analysis End

graph in Fig. 5, a 2-way cache with 2 sets and suppose that the cost difference between a miss and a hit is 10, we can discard in the analysis the execution path represented by the hexagon, because the accumulated time difference between this path and any execution path with a accumulated time of 120 (represented by the triangle or by the square in Fig. 5) is 60 that it is bigger than the cost of refilling the whole cache and therefore, we can guarantee that the above mentioned path will never contribute to the "real" WCET.

By construction the LCDET can be found when all accesses are misses on B (maximum cost) and, *whenever possible*, they are hits on A (minimum cost within the same access sequence).In this case we do not consider a general worst cost difference for each access $(m-h)$ but a concrete one which depends on the cache contents. For every single access c, the possible cases are:

1. c is contained in both A and B; $cost_A = h$, $cost_B = h$, $costDiff_{B-A} = 0$
2. c is neither contained in A nor B; $cost_A = m$, $cost_B = m$, $costDiff_{B-A} = 0$
3. c is not contained in A but it is in B; $cost_A = m$, $cost_B = h$, $costDiff_{B-A} = h - m$
4. c is contained in A but not in B; $cost_A = h$, $cost_B = m$, $costDiff_{B-A} = m - h$

Clearly, case 4 $(m-h)$ is the desired one, since it maximizes the cost difference. However, there can be situations where case 4 is not possible. In these situations, the next worst one is case 2. This case does not increase the cost difference, but produces a miss in B, which follows our worst case construction (all misses in B and hits in A whenever possible). This case 2 is always possible, since it represents accessing a new line.

Thus, we have to look for the worst case on every step. That is, case 4 whenever possible and case 2 otherwise. On each case, cache states A and B are modified accordingly and the total cost is the sum of the cost of each step. At the end, we have constructed an incoming sequence, access by access, guaranteeing that the LCDET cost is minimum.

Algorithm 1 Algorithm for obtaining the *LCDET* from cache state A to B ($LCDET_{B-A}$).

```
Require: $A_{1,\ldots,S}, B_{1,\ldots,S}$: cache states, i.e. array (sets) of ordered lists (ways).
Ensure: costDiff: LCDET from A to B ($LCDET_{B-A}$).
1:  costDiff ← 0
2:  for $s = 1, \ldots, S$ do {$S$-set cache}
3:    for $n = 1, \ldots, N$ do {$N$-way set}
4:      $c \leftarrow$ search $c \in A_s$ / $c \notin B_s$
5:      if $\exists\, c \in A_s$ / $c \notin B_s$ then
6:        costDiff ← costDiff + $(m - h)$
7:        $A_s \leftarrow$ modify cache state $A_s$
8:        $B_s \leftarrow$ modify cache state $B_s$
9:      else
10:       costDiff ← costDiff + 0
11:       $A_s \leftarrow$ modify cache state $A_s$
12:       $B_s \leftarrow$ modify cache state $B_s$
13:     end if
14:   end for
15: end for
16: return costDiff
```

Corollary 4: The lowest LCDET is obtained by algorithm 1.

For example, if we consider the EPAS-graph in Fig. 5, a 2-way cache with 2 sets and suppose that the cost difference between a miss and a hit is 10, applying the algorithm 1 to cache states that Fig. 6 is showing, the LDCET obtained from the cache state in EPAS represented by the pentagon to the cache state in EPAS represented by the triangle is 10, therefore we can discard in the analysis the execution path represented by the pentagon, since we can guarantee that the execution path represented by the pentagon will never be the "real" WCET.

Note that after applying our techniques in order to prune execution paths (*equal-cache path removal* and *arbitrary-cache path removal*) to the code of Fig. 1, the exact WCET requires only to analyze 2 paths instead of the 16 possible.

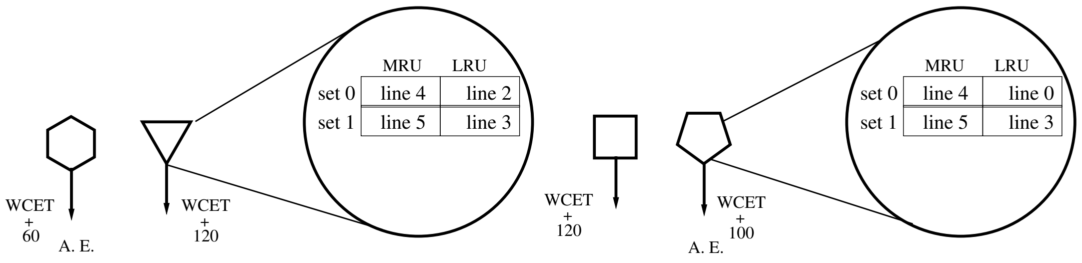


Fig. 6. Path removal with arbitrary cache state. A.E. stands for Analysis End

V. Conclusions and future work

In this paper we propose a new approach for the WCET analysis of a program. Our approach is based on reducing the number of states to analyze without losing information.

Namely, our techniques discard execution paths which are not relevant for the WCET computation. Our first technique is specially effective at loops, where the number of possible execution paths becomes exponential in the number of iterations. In this case, we prove that it suffices the analysis of a much smaller number, dependent only on the number of different paths inside a loop.

Our second technique computes the difference of the accumulated execution time between two execution paths at a given instruction. If this difference is bigger than a threshold (previously computed) we can safely prune the execution path with smaller accumulated execution time.

Thus, by combining the two techniques it is possible to compute a very accurate WCET, instead of obtaining it by overstimated bounds (caused by information loss) as other methods do.

We propose as future work a detailed study of the application of our methods, that is, to analyze the most convenient points to apply each technique and to measure the degree of pruning reached in real workloads. Another interesting research line is the generalization of our approaches to other hardware components with a sequential behavior, such as TLBs or branch predictors.

Referencias

[1] L. C. Aparicio, J. Segarra, J. L. Villarroel, and V. Viñals. Execution path pruning for WCET analysis. Technical Report RR-06-08, DIIS Universidad de Zaragoza, June 2006.

[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 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY, 1977.

[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] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[6] 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.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 3rd edition, 2003.

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

[9] A. Martí, A. Perles, and J. V. Busquets. Static use of locking caches in multitask preemptive real-time systems. In *IEEE Real-Time Embedded System Workshop*, December 2001.

[10] A. Martí, A. Pérez, A. Perles, and J. V. Busquets. Using genetics algoritms in content selection for locking-caches. In *Proceedings of the IAESTED International Conference on Applied Informatics*, 2001.

[11] F. Mueller. Timing predictions for multi-level caches. In *ACM SIGPLAN Workshop on Language, Compiler and Too Support for Real-Time Systems*, pages 29–36, June 1997.

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

[13] 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.

[14] I. Puaut, A. Arnaud, and D. Decotigny. Performance analysis of static cache locking in multitasking hard real-time systems. Technical Report 1568, IRISA, October 2003.

[15] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, December 2002.

[16] H. Ramaprasad and F. Mueller. Bounding worts-case data cache behavior by analytically deriving cache reference patterns. In *Real-Time and Embedded Technology and Applications Symposium*, pages 148–157, May 2005.

[17] 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.

[18] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'03)*, pages 272–282, June 2003.

[19] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 154–166, December 2003.

[20] X. Vera and J. Xue. Let's study whole program cache behaviour analytically. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.

[21] R. White, F. Mueller, C. Healy, D.Whalley, and M. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2-3):209–233, November 1999.

[22] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 192–202, June 1997.