WCET analysis of multi-level non-inclusive set-associative instruction caches *

Damien Hardy Isabelle Puaut
Université Européenne de Bretagne / IRISA, Rennes, France

Abstract

With the advent of increasingly complex hardware in real-time embedded systems (processors with performance enhancing features such as pipelines, cache hierarchy, multiple cores), many processors now have a set-associative L2 cache. Thus, there is a need for considering cache hierarchies when validating the temporal behavior of real-time systems, in particular when estimating tasks' worst-case execution times (WCETs). In this paper, we propose a safe static instruction cache analysis method for multi-level non-inclusive caches. The proposed method is experimented on medium-size and large programs. We show that the method is reasonably tight. We further show that in all cases WCET estimations are much tighter when considering the cache hierarchy than when considering only the L1 cache. An evaluation of the analysis time is conducted, demonstrating that analyzing the cache hierarchy has a reasonable computation time.

1. Introduction

Cache memories are introduced to decrease the access time to the information due to the increasing gap between fast micro-processors and relatively slower main memories. Caches are very efficient at reducing average-case memory latencies for applications with good spatial and temporal locality. Architectures with caches are now commonly used in embedded real-time systems due to the increasing demand for computing power of many embedded applications.

In real-time systems it is crucial to prove that the execution of a task meets its deadline in all execution situations, including the worst-case. This proof needs an estimation of the worst-case execution time (WCET) of any sequential task in the system. WCET estimates have to be safe (larger than or equal to any possible execution time). Moreover, it has to be tight (as close as possible to the actual worst-case execution time) to correctly dimension the resources required by the system.

The presence of caches in real-time systems makes the estimation of both safe and tight WCET bounds difficult due to the dynamic behavior of caches. Safely estimating WCET on architectures with caches requires a knowledge of all possible cache contents in every execution context, and requires some knowledge of the cache replacement policy.

During the last decade, much research has been undertaken to predict WCET in architectures equipped with caches. Regarding instruction caches, static cache analysis methods have been designed, based on the so-called *static cache simulation* [9, 11] or *abstract interpretation* [17, 5]. Approaches for static data cache analysis have also been proposed [16]. Other approaches like cache locking have been suggested when the replacement policy is hard to predict precisely [13] or for data caches [18]. The impact of multi-tasking has also been considered by approaches aiming at statically determining cache related preemption delays [12].

To the best of our knowledge, only [10] deals with cache hierarchies. However, we show that this method can be unsafe for some cache structures and reference streams.

The contribution of this paper is the proposal of a new safe cache analysis method for multi-level non-inclusive set-associative caches. Our approach can be applied to caches with different replacement policies thanks to the reuse of an existing single-level cache analysis method. The safety of the proposed method relies on the introduced concept of *cache access classification* (CAC), defining which references are used for the analysis of every cache level, in conjunction with the more traditional cache hit/miss classification. This paper presents experimental results showing that in most cases the analysis is tight. Furthermore, in all cases WCET estimations are much tighter when considering the cache hierarchy than when considering the L1 cache only. An evaluation of the analysis time is also presented, demonstrating that analyzing the cache hierarchy has a reasonable computation time.

The rest of the paper is organized as follows. Related work is surveyed in Section 2. Section 3 presents the types of caches to which our analysis applies. Section 4 presents a counterexample showing that the approach presented in [10] may produce underestimated WCET estimates when analyzing set-associative caches. Section 5 then details our proposal. Experimental results are given in Section 6. Finally, Section 7 concludes with a summary of the contributions of this paper, and gives directions for future work.

*This study was partially supported by the french National Research Agency project Mascotte (ANR-05-PDIT-018-01)

2. Related work

Caches in real-time systems raise timing predictability issues due to their dynamic behavior and their replacement policy. Many static analysis methods have been proposed in order to produce a safe WCET estimate on architectures with caches. To be safe, existing static cache analysis methods determine *every* possible cache contents at every point in the execution, considering all execution paths altogether. Possible cache contents can be represented as sets of *concrete cache states* [12] or by a more compact representation called *abstract cache states (ACS)* [17, 5, 10, 11].

Two main classes of approaches [17, 11] exist for static WCET analysis on architectures with L1 caches. In [17] the approach is based on *abstract interpretation* [4] and uses ACS. In this approach, three different analyses are applied which use fixpoint computation to determine: if a memory block is *always* present in the cache (*Must* analysis), if a memory block *may* be present in the cache (*May* analysis), or if a memory block will not be evicted after it has been first loaded (*Persistence* analysis). A *cache categorization* (e.g. *always-hit*, *first-miss*) can then be assigned to every instruction based on the results of the three analyses. This approach originally designed for LRU set-associative caches has been extended for different cache replacement policies in [6]. To our knowledge, this approach has not been extended to analyze multiple levels of caches. Our multi-level non-inclusive cache analysis will be defined using [17], mainly because of the theoretical results applicable when using abstract interpretation. In [9], *static cache simulation* is used to determine every possible content of the cache before each instruction. Static cache simulation computes abstract cache states using dataflow analysis. A *cache categorization* (*always-hit*, *always-miss*, *first-hit* and *first-miss*) is used to classify the worst-case behavior of the cache for a given instruction. The base approach, initially designed for direct-mapped caches, was later extended to set-associative caches in [11].

The cache analysis method presented in [9] has also been extended to cache hierarchies in [10]. A separate analysis of each memory level is performed by first analyzing the behavior of the L1 cache. The result of the analysis of the L1 cache is consequently used as an input to the analysis of L2 cache, and so on. The approach considers an access to the next level of the memory hierarchy (e.g. L2 cache) if the access is not classified as *always-hit* in the current level (e.g. L1 cache). As shown in Section 4, this filtering of memory accesses, although looking correct at the first glance, is unsafe for set-associative caches. Our work is based on the same principles as [10] (cache analysis for every level of the memory hierarchy, filtering of memory accesses), except that the unsafe behavior present in [10] is removed thanks to the introduction of the concept of *cache access classification* (CAC), defining which references are used for the analysis of every cache level.

3. Assumptions

We consider a hierarchy of N levels of instruction caches, level 1 representing the internal cache (L1 cache). Every cache is set-associative. Although both our examples and our performance evaluation apply to caches with a Least Recently Used (LRU) replacement policy, the proposed multi-level non-inclusive cache analysis is not tied to a specific cache replacement policy. There are no constraints on the cache line sizes for the different cache levels. It is further assumed that the following three properties hold:

P1. A piece of information is searched for in the cache of level L if and only if a cache miss occurred when searching it in the cache of level $L-1$. Cache of level 1 is always accessed.

P2. Every time a cache miss occurs at cache level L, the entire cache line containing the missing piece of information is always loaded into the cache of level L.

P3. There are no actions on the cache contents (i.e. lookups/modifications) other than the ones mentioned above.

Property P1 rules out architectures where cache levels are accessed in parallel to speed up the search for a piece of information. Property P2 rules out architectures with exclusive caches. Finally, property P3 implies that there are no actions like the invalidation of a cache line or the update of the age at a given level of the cache hierarchy to provide inclusive caches. Our method thus applies to non-inclusive caches.

Remark that when only P2 and P3 hold and P1 is relaxed (i.e. every access is always propagated to all cache levels), the different cache levels are then independent. Consequently, there is no need to define any specific cache analysis technique; existing single-level static cache analysis like [17, 9] can be applied to each level in isolation.

Although not explicitly mentioned in [10], their multi-level analysis relies on properties P1, P2 and P3 and thus does not apply to inclusive caches. Property P1 is assumed because in [10] the different levels of caches are analyzed sequentially, with a filtering of hits between the successive cache levels. Property P2 is assumed because all non-filtered references result in a modification of the cache state of the next cache level, due to the use of *static cache simulation*. Finally, property P3 holds because there is no mention of any action on the cache contents other than the ones considered in properties P1 and P2. This is not enough, as shown in [1] to ensure cache inclusion.

4. Limitation of multi-level cache analysis [10]

The static cache simulation method presented in [11] is defined for a single level of cache. It computes abstract cache states (ACS) using dataflow analysis. The output of the analysis is a classification of each memory references as *first-miss*,

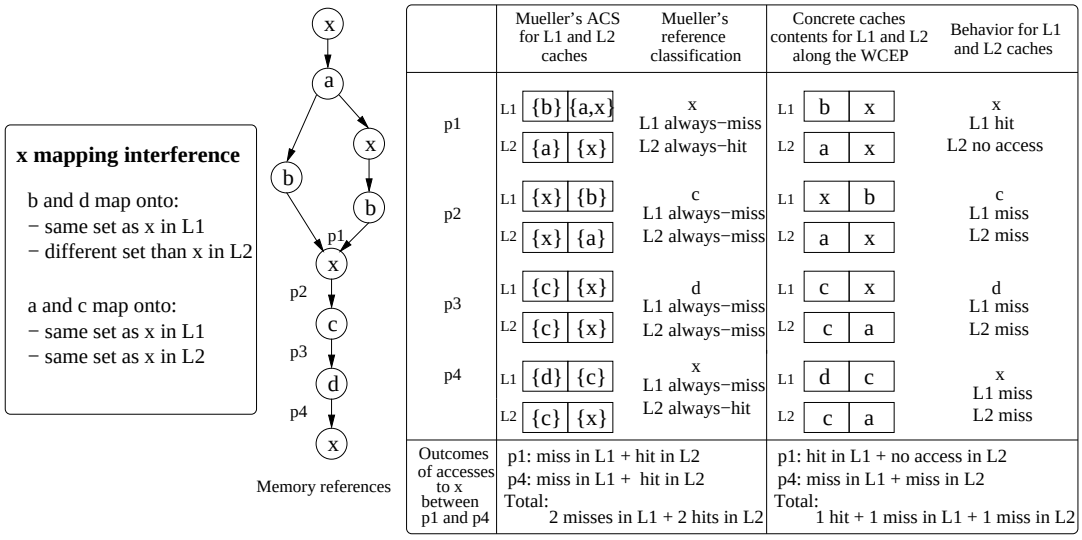


Figure 1. Example of limitation for 2-way L1 and L2 non-inclusive caches

first-hit, *always-miss*, or *always-hit*. An *always-hit* means that the reference is guaranteed to be in the cache and an *always-miss* is used when a reference is not guaranteed to be in the cache (but may be in the cache for some execution paths). Categories *first-hit* and *first-miss* are used for references enclosed in loops[1]. To classify a reference r as an *always-hit*, the single-level cache analysis method of [11] uses ACS and *dominator cache state* (DCS). The ACS is used to determine the number of references in conflict with r in a set. DCS is used to determine if reference r *must* be cached due to an earlier reference in *all* the possible execution paths. If the number of references in conflict with r is strictly lower than the degree of associativity and r is present in the DCS, the access is classified as an *always-hit*.

In [10], this approach is extended to a multi-level cache hierarchy. The analysis performs a separate and sequential analysis for each level in the memory hierarchy. The output of the analysis for level L is the classification of each memory references, subsequently used as an input for the analysis of level $L + 1$. All references are considered when analyzing level $L + 1$ except those classified as *always-hit* at level L (or at a previous level). The implicit assumption behind this filtering of memory accesses is that when it cannot be guaranteed that a reference is a hit at level L, the worst-case situation occurs when a cache access to level $L + 1$ is performed. Unfortunately, this assumption is not safe as soon as the degree of associativity is greater than or equal to two, as shown on the counterexample depicted in Figure 1.

The figure represents possible streams of memory references on a system with a L1 2-way associative cache and a L2 2-way associative cache, both with a LRU replacement policy. The safety problem is observed on reference x, assumed to be performed inside a function. References a, b, c, and d do not cause any safety problem (they cause misses in the L1 and L2 both at analysis time and at run-time); they are introduced only to illustrate the safety problem on reference x. Let us assume that:

- b and d map onto the same set as x in the L1 cache and map onto a different set than x in the L2 cache. This case is likely to occur because the size of the L1 cache is smaller than the size of the L2 cache.
- a and c map onto the same set as x in the L1 cache and in the L2 cache.

The left part of the figure presents the contents of the ACS at points $p1$, $p2$, $p3$ and $p4$ in the reference stream. For the sake of conciseness, only the set of the ACS where reference x is mapped is shown, as well as the the resulting classification. In the figure, $\boxed{\{u\} \mid \{v, w\}}$ represents the possible contents of the two cache lines of the set at each point, the left cache line has an age lower than the right one; $\{v, w\}$ means that both v and w may be in the cache line. The right part of the figure presents the concrete cache contents at the same points when the worst-case execution path (WCEP), which takes the right path in the conditional construct, is followed.

The ACS of the L2 cache at $p1$ is $\{a\}, \{x\}$. This due to the filtering of reference x along the right path of the conditional (x is classified as an *always-hit* in the L1 cache and thus is not propagated to the L2 cache). The access to x at $p1$ produces an *always-miss* in the L1 cache (the number of references in conflict with x is equal to the degree of associativity). Consequently, this access is propagated to the L2 cache. With this process, reference x is present in the DCS of the L2 cache at point $p4$ and the number of references in conflict with x is then strictly lower than the degree of associativity. So, the x reference at $p4$ is classified as an *always-hit* in the L2 cache.

[1]These categories will not be detailed because they are not required to highlight the safety issue of [10].

From the classification of reference x, the analysis outcome between $p1$ and $p4$ is 2 misses in the L1 cache + 2 hits in the L2 cache. In contrast, executing the worst-case reference stream results in 1 hit in the L1 cache + 1 miss in the L1 cache + 1 miss in the L2 cache. Assuming an architecture where a miss is the worst-case and $2 * Thit_{L2} < Tmiss_{L2}$, the contribution to the WCET of the cache accesses to x when executing the code is larger than the one considered in the analysis, which is not safe. This counterexample has been coded, in order to check that this counter-intuitive behavior actually occurs in practice. The safety problem is due to the combination of several factors: (i) the reference stream characteristics, (ii) considering uncertain accesses as misses, (iii) considering an access to the next level in such cases.

To further explain the reasons of the safety problem, let us define, assuming a LRU replacement policy, the *set reuse distance* between two references to the same memory block for a cache level L as the position in the set (equivalent to its way) of the memory block when the second reference occurs. If the memory block is not present when the block is referenced for the second time then the set reuse distance is greater than the number of ways. For instance, the set reuse distance of x on Figure 1 at point $p4$ for analysis [10] is 3 in the L1 cache (greater than the number of L1 ways) and 2 in the L2 cache (present in the second way). In contrast for the possible concrete cache this value is 3 (not present in L1 cache) and 3 (not present in L2 cache). In [10], uncertain accesses are *always* propagated to the next cache level and the analysis may thus underestimate the set reuse distance. This underestimation then results in more hits in the next level in the analysis than in a worst-case execution. Our approach fixes the problem by enumerating the two possible behaviors of every uncertain access (i.e. considering that the access may occur or not).

5. WCET analysis of multi-level non-inclusive caches

After a brief overview of the structure of our multi-level non-inclusive cache analysis framework (§ 5.1), we define in this section the classification of memory accesses (§ 5.2), and detail the analysis and prove its termination (§ 5.3). The use of the cache analysis outputs for WCET computation is presented in § 5.4.

5.1. Overview

Our static multi-level non-inclusive set-associative instruction cache analysis is applied to each level of the cache hierarchy separately. The approach analyzes the first cache level (L1 cache) to classify every reference according to its worst-case cache behavior (*always-hit, always-miss, first-hit, first-miss* and *not classified*, see § 5.2). However, this cache hit/miss classification ($CHMC$) is not sufficient to know if an access to a memory block may occur at the next cache level (L2). Thus, a *cache access classification (CAC)* (*Always*, *Never* and *Uncertain*, see § 5.2) is introduced to capture if it can be guaranteed that the next cache level will be accessed or not.

The combination of the CHMC and the CAC at a given level is used as an input of the analysis of the next cache level in the memory hierarchy. Once all the cache levels have been analyzed, the cache classification of each level is used to estimate the WCET. This framework is illustrated in Figure 2.

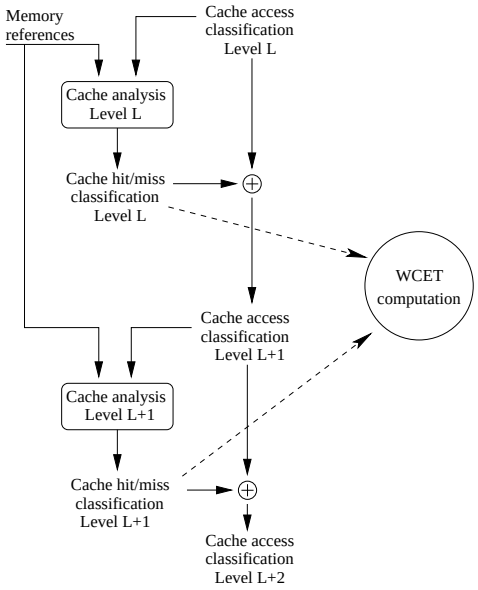


Figure 2. Multi-level non-inclusive cache analysis framework

5.2. Cache classification

Cache hit/miss classification. Due to the semantic variation of the cache classification between static cache simulation [11] and abstract interpretation [17] approaches, we detail the cache hit/miss classification (CHMC) used in our analysis, similar to the one used in [17]:

- *always-hit* (AH): the reference is guaranteed to be in cache,
- *always-miss* (AM): the reference is guaranteed not to be in cache,
- *first-hit* (FH): the reference is guaranteed to be in cache the first time it is accessed, but is not guaranteed afterwards,
- *first-miss* (FM): the reference is not guaranteed to be in cache the first time it is accessed, but is guaranteed afterwards,
- *not-classified* (NC): the reference is not guaranteed to be in cache and is not guaranteed not to be in cache.

Cache access classification. In order to know if an access to a memory block may occur at a given cache level, we introduce a *cache access classification* (CAC). It is used as an input of the cache analysis of each level to decide if the block has to be considered by the analysis or not. The cache access category for a reference r at a cache level L is defined as follows:

- N (Never): the access to r is never performed at cache level L,
- A (Always): the access to r is always performed at cache level L,
- U (Uncertain): it cannot be guaranteed that the access to r is always performed or is never performed at level L.

The cache access classification for a reference r at a cache level L depends on the results of the cache analysis of the reference r at the level $L-1$ (cache hit/miss classification, and cache access classification). The CAC for a reference r at level L is N (never) when the cache hit/miss classification for r at a previous level is *always-hit* (i.e. it is guaranteed that accessing r will never require an access to cache level L). On the other side, the CAC for a reference r at level L is A for the first level of the cache hierarchy, or when CHMC and CAC at level $L-1$ are respectively *always-miss* and A (i.e. it is guaranteed that accessing will always require an access to cache level L). The CAC for reference r at level L is U in all the other cases, expressing the uncertainty that the cache level L is accessed. As later detailed in § 5.3, the cache analysis for U accesses explores the two cases where r accesses cache level L or not, to identify the worst-case.

Table 1 shows all the possible cases of cache access classifications for cache level L depending on the results of the analysis of level $L-1$ (CACs and CHMCs).

$CAC_{r,L-1}$ \ $CHMC_{r,L-1}$	AM	AH	FH	FM	NC
A	A	N	U	U	U
U	U	N	U	U	U
N	N	N	N	N	N

Table 1. Cache access classification for level L ($CAC_{r,L}$)

The contents of the table motivates the need of the cache access classification. Indeed, in case of an *always-miss* at level $L-1$, determining if a reference r should be considered at level L requires more knowledge than the CHMC can provide: if r is always referenced at level $L-1$ ($CAC_{r,L-1} = A$), it should also be considered at level L; similarly, if it is unsure that r is referenced at level $L-1$ ($CAC_{r,L-1} = U$), the reference is still unsure at level L.

It also has to be noted that in the case of an N access, the cache hit/miss classification can be disregarded because the value will be ignored during the WCET computation step for the considered level.

5.3. Multi-level analysis

The proposed multi-level analysis is based on a well known single-level cache analysis method [17]. The analysis presented in [17] is used due to the theoretical results of abstract interpretation [4], and the support for multiple replacement policies [17, 6] (LRU, Pseudo-LRU, Pseudo-Round-Robin). Nevertheless, our analysis can also be integrated into the static cache simulation method [11].

The method detailed in [17] is based on three separate fixpoint analyses applied on the program control flow graph:

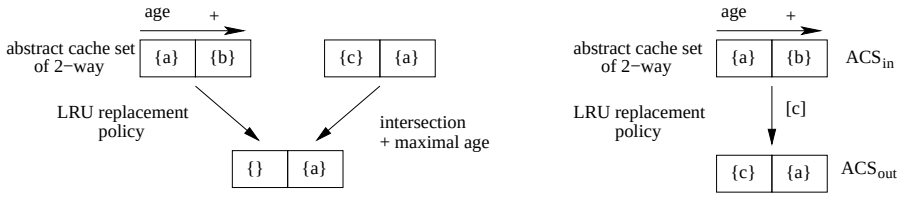
- a *Must* analysis determines if a memory block is always present in the cache at a given point: if so, the block CHMC is *always-hit*;
- a *May* analysis determines if a memory block may be in the cache at a given point: if not, the block CHMC is *always-miss*. Otherwise, if present neither in the Must analysis nor in the Persistence analysis the block CHMC is *not classified*;
- a *Persistence* analysis determines if a memory block will not be evicted after it has been loaded; the CHMC of such blocks is *first-miss*.

Abstract cache states are computed at every basic block. Two functions on the abstract domain, named $Update$, and $Join$ are defined for each analysis:

- Function $Update$ is called for every memory reference on an ACS to compute the new ACS resulting from the memory reference. This function considers both the cache replacement policy and the semantics of the analysis.
- Function $Join$ is used to merge two different abstract cache states when a basic block has two predecessors in the control flow graph, like for example at the end of a conditional construct.

Figure 3 gives an example of the $Join$ (3.a) and $Update$ (3.b) functions for the *Must* analysis for a 2-way set-associative cache with LRU replacement policy. As in this context sets are independent from each other, only one set is depicted. A concept of *age* is associated with the cache block of the same set. The smaller the block age the more recent the access to the block. For the *Must* analysis, memory block a is stored only once in the ACS, with its maximum age. It means that its actual age at run-time will always be lower than or equal to its age in the ACS. The $Join$ and $Update$ functions are defined as follows for the *Must* analysis with LRU replacement (see Figure 3):

- The $Join$ function applied to two ACS results in an ACS containing only the references present in the two input ACS and with their *maximal* age.
- The $Update$ function performs an access to a memory reference c using an input abstract cache state ACS_{in} (the abstract cache state before the memory access) and



a. Join function of Must analysis

b. Update function of Must analysis

Figure 3. $Join$ **and** $Update$ **functions for the Must analysis with LRU replacement**

produces an output abstract cache state ACS_{out} (the abstract cache state after the memory access). The $Update$ function maps c onto its ACS_{out} set with the younger age and increases the age of the other memory blocks present in the same set in ACS_{in}. When the age of a memory block is higher than the number of ways, the memory block is evicted from ACS_{out}.

For the other analyses (*May* and *Persistence*), the approach is similar and the $Join$ function is defined as follows:

- *May* analysis: union of references present in the ACS and with their *minimal* age;
- *Persistence* analysis: union of references present in the ACS and with their *maximal* age.

For more details see [17] and for the other replacement policies see [6].

Extending the single-analysis of [17] to multi-level non-inclusive caches requires a re-definition of the base function $Update$ to take into account the uncertainty of some references at a given cache level, expressed by the cache access classifications (CAC). Function $Join$ needs not to be modified. Function $Update$ (named hereafter $Update_m$ to distinguish our function from the original one) is defined as follows, depending on the CAC of the currently analyzed reference r:

- **A (Always) access.** In the case of an A access the original $Update$ function is used.

  $ACS_{out} = Update(ACS_{in}, r)$; $Update_m \Leftrightarrow Update$

- **N (Never) access.** In the case of an N access, the analysis does not consider this access at the current cache level, so the abstract cache state stays unchanged.

  $ACS_{out} = ACS_{in}$; $Update_m \Leftrightarrow identity$

- **U (Uncertain) access.** In the case of a U access, the analysis deals with the uncertainty of the access by considering the two possible alternative sub-cases (see Figure 4 for an illustration):
  - the access is performed. The result is then the same as an A access;
  - the access is not performed. The result is then the same as a N access.

  To obtain the ACS_{out} produced by a U access, we merge this two different abstract cache states by the $Join$ function.

$$ACS_{out} = Join(Update(ACS_{in}, r), ACS_{in})$$
$$Update_m(ACS_{in}, r) = Join(Update(ACS_{in}, r), ACS_{in})$$

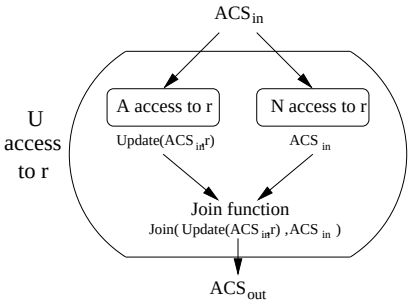


Figure 4. $Update_m$ **function for U access**

The original functions $Join$ and $Update$ produce a safe hit/miss classification of the memory references. In our case, this validity is kept for the A accesses and is obvious for the N accesses. As for the U accesses, which are the key to ensure safety, the analyses have to keep the semantics of each analysis. For the *Must* and *Persistence* analyses, the $Update_m$ function maintains the maximal age of each memory reference by the original $Join$ function applied to the two ACS (access occurs or not). Similarly, for the *May* analysis, the minimal age is kept by the $Update_m$ function. So the semantic of each analysis is maintained by the $Update_m$ function.

Termination of the analysis. It is demonstrated in [17] that the domain of abstract cache states is finite and, moreover, that the $Join$ and $Update$ functions are monotonic. So, using ascending chains (every ascending chain is finite) proves the termination of the fixpoint computation.

In our case, the only modification to [17] is the $Update$ function. Thus, to prove the termination of our analysis we have to prove that the modified function $Update_m$ is monotonic for each type of cache access.

Proof: for an A access, $Update_m$ is identical to $Update$, so it is monotonic. For an N access $Update_m$ is the identity function, so it is monotonic. Finally, for a U access, $Update_m$ is a composition of $Update$ and $Join$. As the composition

of monotonic functions is monotonic, $Update_m$ is then also monotonic. This guarantees the termination of our analysis for each type of cache access and thus for the whole analysis. □

It is important to note that our analysis terminates for any monotonic $Update/Join$ functions. Thus, all $Update/Join$ functions defined in [17, 6] to model different replacement policies can be directly reused.

5.4. WCET computation

The result of the multi-level analysis gives the worst-case access time of each memory reference to the memory hierarchy. In other words, this analysis produces the contribution to the WCET of each memory reference, which can be included in well-known WCET computation methods [15, 14].

In the formulae given below, the contribution to the WCET of a NC reference at level L is the latency of an access to level L+1, which is safe for architectures without timing anomalies caused by interactions between caches and pipelines, as defined in [8]. For architectures with such timing anomalies (e.g. architectures with out-of-order pipelines), more complex methods such as [7] have to be used to cope with the complex interactions between caches and pipelines.

We define the following notations: constant $Thit_\ell$ represents the cost in cycles of a hit at level ℓ (accesses to the main memory are always hits), $first$ and $next$ to distinguish the first and the successive execution in loops, the binary variables $first\_present_\ell(r)$ and $next\_present_\ell(r)$ represent that an access to reference r occurs (1) or not (0) at level ℓ. Finally, the sum of variables $COST\_first(r)$ and $COST\_next(r)$ give the contribution to the WCET of a reference r at a given point in the program, that can be used to compute the WCET. $COST\_first(r)$ and $COST\_next(r)$ are computed as follows:

$$COST\_first(r) = \sum_{\ell=1}^{n} Thit_\ell * first\_present_\ell(r)$$

$$COST\_next(r) = \sum_{\ell=1}^{n} Thit_\ell * next\_present_\ell(r)$$

$first\_present_\ell(r)$ and $next\_present_\ell(r)$ are defined as follows:

$$first\_present_\ell = \begin{cases} 1 \text{ if } & \ell = 1 \\ 1 \text{ if } & first\_present_{\ell-1} = 1 \\ & \wedge\ (CHMC_{\ell-1} = AM \\ & \quad \vee CHMC_{\ell-1} = FM \\ & \quad \vee CHMC_{\ell-1} = NC) \\ 0 & \text{otherwise} \end{cases}$$

$$next\_present_\ell = \begin{cases} 1 \text{ if } & \ell = 1 \\ 1 \text{ if } & next\_present_{\ell-1} = 1 \\ & \wedge\ (CHMC_{\ell-1} = AM \\ & \quad \vee CHMC_{\ell-1} = FH \\ & \quad \vee CHMC_{\ell-1} = NC) \\ 0 & \text{otherwise} \end{cases}$$

6. Experimental results

In this section, we evaluate the tightness of our static multi-level non-inclusive cache analysis comparatively to the execution in a worst-case scenario. We also evaluate the extra computation time caused by the analysis of the cache hierarchy. We first describe the experimental conditions and then we give and analyze experimental results for 2-level and 3-level cache hierarchies.

6.1. Experimental setup

Cache analysis and WCET estimation. The experiments were conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 with no optimization. The WCETs of tasks are computed by the Heptane[2] timing analyzer [3], more precisely its Implicit Path Enumeration Technique (IPET[3]). The *Must*, *May* and *Persistence* analyses are conducted sequentially on every level of the cache hierarchy, all caches implementing a LRU replacement policy. The analysis is context sensitive (function are analyzed in each different calling context).

To separate the effect of the caches from those of the parts of the processor micro-architecture, WCET estimation only takes into account the contribution of caches to the WCET as presented in Section 5.4. The effects of other architectural features are not considered. In particular, we do not take into account timing anomalies caused by interactions between caches and pipelines, as defined in [8]. The cache classification *not-classified* is thus assumed to have the same worst-case behavior as *always-miss* during the WCET computation in our experiments. The cache analysis starts with an empty cache state.

The computation time measurement is realized on an Intel Pentium 4 3.6 GHz with 2 GB of RAM.

Measurement environment. The measure of the cache activities on a worst-case execution scenario uses the Nachos educational operating system[4], running on top of a simulated MIPS processor. We have extended Nachos with a three-level cache hierarchy with a LRU replacement policy at each level.

Benchmarks. The experiments were conducted on five small benchmarks and two tasks from a larger real application (see Table 2 for the application characteristics). All small benchmarks are benchmarks maintained by Mälardalen WCET research group[5]. The real tasks are part of the case study provided by the automotive industrial partner of the Mascotte ANR project[6] to the project partners.

[2] Heptane is an open-source static WCET analysis tool available at *http://www.irisa.fr/aces/software/software.html*.

[3] So-called IPET methods estimate WCET by solving linear equations generated from the program control flow graph [19].

[4] Nachos web site, http://www.cs.washington.edu/homes/tom/nachos/

[5] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[6] http://www.projet-mascotte.org/

Name	Description	Code size (bytes)
matmult	Multiplication of two 50x50 integer matrices	1200
ns	Search in a multi-dimensional array	600
bs	Binary search for the array of 15 integer elements	336
minver	Inversion of floating point 3x3 matrix	4408
jfdctint	Integer implementation of the forward DCT (Discrete Cosine Transform)	3040
adpcm	Adaptive pulse code modulation algorithm	7740
task1	Confidential	12711
task2	Confidential	12395

Table 2. Benchmark characteristics

6.2. Results for a 2-level hierarchy

Precision of the multi-level analysis. In order to determine the tightness of the multi-level analysis, static analysis results are compared with those obtained by executing the programs in their worse-case scenario. Due to the difficulty in identifying the input data that results in the worst-case situation in complex programs, we only use the simplest benchmarks (*matmult, ns, bs, minver, jfdctint*) to evaluate the precision of the analysis. All benchmarks but *bs* are single-path programs, and *bs* is simple enough to make the identification of its worst-case input data obvious.

Small L1 and L2 instruction caches are used in this part of the performance evaluation in order that the code of most of the benchmarks (except *ns* and *bs*) do not fit into the caches. The L1 cache is 1KB large, 4-way associative with 32B lines. We use two different L2 caches configurations of 2KB 8-way associative: one with 64B lines and another one with 32B lines.

To evaluate the precision of our approach, the comparison of the hit ratio at the L2 level between static analysis and measurement is not appropriate. Indeed, the inherent pessimism of the static cache analysis at the L1 level introduces some accesses at the L2 level that never happen at run-time. Instead, the results are given in Table 3 using two classes of metrics, given for each benchmark in separate rows. For each cache configuration, the values of these metrics are given both for predicted values (left column) and measured values (right column).

- The number of references and the number of misses at every level of the memory hierarchy (top row) to show the behavior of the multi-level non-inclusive cache analysis.
- The contribution of the memory accesses to the WCET (bottom row). Two predicted values are given: one considering a cache hierarchy (L1+L2) and one when ignoring the L2 cache (L1 only) to demonstrate the usefulness of the multi-level analysis. To compute it, we use a L1 hit cost of 1 cycle, a L2 hit cost of 10 cycles and a memory latency of 100 cycles. When considering only one cache level, the memory latency is 110 cycles. A single measured value is given in the right column for a cache configuration with both a L1 and a L2 cache.

Two types of behaviors can be observed depending on the application structure:

- The first type of situations is when the number of L1 misses computed statically is very close to the measured value (benchmark *jfdctint*). In this benchmark, the base cache analysis applied to the L1 cache is very tight due to the application structure (presence of big basic blocks and a small number of control structures). As a consequence, the reference stream considered during the analysis of the L2 cache is very close to the accesses actually performed at run-time. Thus, the number of misses in the L2 is also very close to the number of L2 misses occurring during execution and the overestimation of the computed cache contribution to the WCET is 2%[7]. In this case, the overall difference between static analysis and execution is mainly due to the pessimism introduced by considering the cache hierarchy (classification as U of every access that cannot be guaranteed to be or not to be in the L1).
- The second type of situations occurs when the static cache analysis at L1 level is slightly less tight (smaller basic blocks and a larger number of control structures). Then, this behavior is also present at the L2 level and it is increased by the introduction of the U accesses. In this case, the multi-level analysis is still reasonably tight: 8% in average using *minver*, *matmult* and *ns*. The only case where the analysis is not tight occurs with *bs* (71%). This is due to the classification as *first-miss* of the accesses performed inside the application loop, combined with the low number of iterations of the loop (4). This behavior highlights the pessimism of the single-level cache analysis for some applications.
  Nevertheless, it turns out that a lot of accesses, not detected as hits by the L1 analysis, can be detected as hits by the L2 analysis. The resulting WCET is thus much smaller than if only one level of cache was considered.

Finally, we do not distinguish different behaviors when the applications fit into the L1 cache (*ns*), does not fit into the L2 cache (*minver*) or fits in the L2 cache but not in the L1 cache (*matmult*).

For the largest codes (*adpcm*, *task1*, *task2*), only results of static cache analysis are given (measurements are not realized due to the difficulties to execute these tasks in their worst-case execution scenario). Since the code size of these three tasks is larger than the one of the simple benchmarks, the cache size is chosen larger and more realistic than the one considered be-

[7] The overestimation is an average of the two considered cache configurations, the overestimation for a configuration being defined as: $(\frac{StaticCacheContribution\_L1+L2}{MeasuredCacheContribution} - 1) * 100$

Benchmark	Metrics	Static Analysis 32B lines for L1 64B lines for L2	Measurement 32B lines for L1 64B lines for L2	Static Analysis 32B lines for L1 32B lines for L2	Measurement 32B lines for L1 32B lines for L2
jfdctint	nb of L1 accesses	8039	8039	8039	8039
	nb of L1 misses	725	723	725	723
	nb of L2 misses	54	49	101	96
	cache contribution to WCET				
	L1+L2, cycles	20689	20169	25389	24869
	L1 only, cycles	87789		87789	
bs	nb of L1 accesses	196	196	196	196
	nb of L1 misses	16	11	16	11
	nb of L2 misses	15	6	16	11
	cache contribution to WCET				
	L1+L2, cycles	1856	906	1956	1406
	L1 only, cycles	1956		1956	
minver	nb of L1 accesses	4146	4146	4146	4146
	nb of L1 misses	150	140	150	140
	nb of L2 misses	108	71	150	140
	cache contribution to WCET				
	L1+L2, cycles	16446	12646	20646	19546
	L1 only, cycles	20646		20646	
ns	nb of L1 accesses	26428	26411	26428	26411
	nb of L1 misses	23	13	23	13
	nb of L2 misses	20	7	23	13
	cache contribution to WCET				
	L1+L2, cycles	28658	27241	28958	27841
	L1 only, cycles	28958		28958	
matmult	nb of L1 accesses	525894	525894	525894	525894
	nb of L1 misses	51	41	51	41
	nb of L2 misses	49	19	51	38
	cache contribution to WCET				
	L1+L2, cycles	531304	528204	531504	530104
	L1 only, cycles	531504		531504	

Benchmark	Metrics	Static Analysis 32B lines for L1 64B lines for L2	Static Analysis 32B lines for L1 32B lines for L2
adpcm	nb of L1 accesses	187312	187312
	nb of L1 misses	2891	2891
	nb of L2 misses	289	297
	cache contribution to WCET		
	L1+L2, cycles	245122	245922
	L1 only, cycles	505322	505322
task1	nb of L1 accesses	1872522	1872522
	nb of L1 misses	678	678
	nb of L2 misses	662	678
	cache contribution to WCET		
	L1+L2, cycles	1945502	1947102
	L1 only, cycles	1947102	1947102
task2	nb of L1 accesses	6783	6493
	nb of L1 misses	792	796
	nb of L2 misses	718	796
	cache contribution to WCET		
	L1+L2, cycles	86503	94053
	L1 only, cycles	93903	94053

Table 3. Precision of the static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache. Cache sizes of 1KB (resp. 2KB) for L1 (resp. L2) in top table, 8KB (resp. 64KB) for L1 (resp. L2) in bottom table).

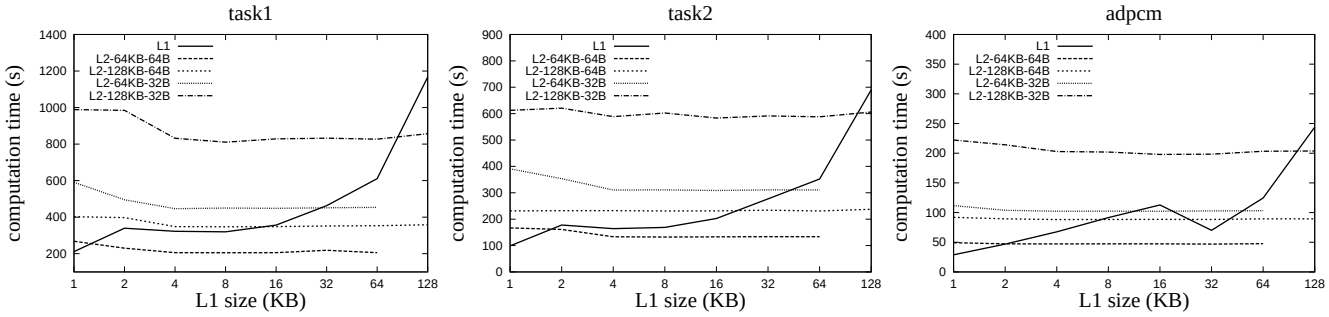


Figure 5. Computation time with a 64KB and a 128KB L2 cache

fore. We use a 8KB large L1 cache and a 64KB large L2 cache with the same cache line size and associativity as before.

We can notice the rather low number of cache hits in the L2 cache in the configuration with 32B L2 cache lines. This is explained by the size of loops in the applications as compared to the L1 cache size. In all tasks, the code of the loops entirely fits into the L1 cache and thus there is no reuse once a piece of code gets loaded into the L2 cache. When the cache line size in the L2 cache is larger, the number of hits in the L2 cache increases significantly, due to the spatial locality of applications.

In summary, the overall tightness of the multi-level non-inclusive cache analysis is strongly dependent on the initial cache analysis of [17]. In all the cases: (i) the extra pessimism caused by our multi-level analysis for the sake of safety (introduction of U accesses) is reasonable, (ii) considering the cache hierarchy generally results in much lower WCETs comparatively to considering only one cache level and an access to main memory for each miss.

Computation time evaluation. The analysis time is evaluated on a two-level cache hierarchy, using the three largest codes (*adpcm*, *task1*, and *task2*) and the same cache structures as before.

What we wish to evaluate is the extra-cost for analyzing the second level of cache comparatively to a traditional cache analysis of only one level. The extra-analysis time mainly depends on the number of references considered when analyzing the L2 cache, which itself depends on the size of the L1 cache (the larger the L1, the higher the number of references detected as hits in the L1 and thus the lower the number of references considered in the analysis of the L2). Thus, we vary the size of the L1 (4-way and cache lines of 32B) from 1KB to L2 cache size.

Figure 5 details the results for 64 KB (32B and 64B line) and 128 KB (32B and 64B line) L2 caches respectively. The X axis gives the L1 cache size in KB. The Y axis reports the computation time in seconds.

The shape of the curves are very similar for each used benchmark and each L2 cache size tested. The computation time for analyzing the L1 cache increases with the size because of the inherent dependency of single-level cache analysis to the cache size. However, the computation time increase is not always monotonic, like for instance for benchmark *adpcm*. This non-monotonic behavior comes from a variation of the number of iterations in the fixpoint computation present in the single-level cache analysis. In contrast, the analysis time of the L2 cache decreases when the L1 cache is increased: as the L1 cache filters more and more memory references, the number of accesses to the L2 cache considered in the analysis is reduced (more and more accesses become N access).

The proposed multi-level non-inclusive cache analysis introduced an extra computation cost for U accesses to explore the two possible behavior of uncertain accesses. It can be observed that this extra cost is not visible because it is masked by the filtering of accesses.

When the L2 cache size is 128 KB the slope of the L2 curve is lower than for a 64 KB cache. This is due to the incompressible time needed for single-level cache analysis of the L2 cache, dependent on the L2 cache size, which masks the filtering effect of the L1 cache. Nevertheless even in this case the computation time is reasonable.

To conclude, the computation time required for the multi-level set-associative non-inclusive cache (L1 + L2) analysis is significant but stays reasonable on the case study application.

6.3. Results for a 3-level hierarchy

We now evaluate the precision of our analysis with a 3-level cache hierarchy. The benchmark used for this experimentation is an enclosing concatenation of the small benchmarks into a loop with a number of iterations of two. This concatenation aggregates the different types of code structures that existed in the small benchmarks (control code with small basic blocks, computation-intensive code with larger basic blocks) into the same benchmark.

The L3 cache is a 16-way associative with 32B lines and a latency of 30 cycles. The analysis is experimented on two sizes of L3 caches: 4KB and 16KB such that the benchmark fits into the L3 cache in its largest configuration and not in its smallest one.

The results are presented in Table 4. The outer loop of this benchmark has a code larger than the L2 cache size which decreases the precision of the persistence analysis of the L1 and

Size of L3 cache	Metrics	Static Analysis 32B lines for L1 32B lines for L2,L3	Measurement 32B lines for L1 32B lines for L2, L3
4 KB	# L1 accesses	1129430	1129425
	# L1 misses	8669	1852
	# L2 misses	5236	608
	# L3 misses	1217	599
16 KB	# L1 accesses	1129430	1129425
	# L1 misses	8669	1852
	# L2 misses	5236	608
	# L3 misses	348	298

Table 4. Precision of the static multi-level analysis (1KB 4-way L1 cache, 2KB 8-way L2 cache and 16-way L3 cache).

L2 caches due to the presence of deeply nested loops. This effect was identified and solved by a multi-loop-level persistence analysis in [2]. Nevertheless, the precision of the L3 cache analysis has a behavior similar to the second case identified in Section 6.2 (small basic blocks and a large number of control structures).

7. Conclusion

In this paper, we have proposed a solution to produce safe WCET estimates of a hierarchy of set-associative instruction caches, whatever the degree of associativity and the cache replacement policy. The safety of the proposed method relies on the introduction of the concept of a cache access classification in conjunction with a cache hit/miss classification. We have proven the termination of the analysis. Moreover, the experimental results show that this method is precise in many cases, generally tighter than considering only one cache level, and has a reasonable computation time on the case study. In future research we will consider data caches and unified L2 caches, by using for instance partitioning techniques to separate instruction from data in the L2 cache. We will also extend this approach to analyze cache hierarchies of multicore architectures, or to other configurations of cache hierarchies (e.g. exclusive caches).

Acknowledgments. The authors are grateful to André Seznec, Robert Guziolowski and to the anonymous reviewers for feedback on earlier versions of the paper.

References

[1] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *ISCA'88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[2] C. Ballabriga and H. Cassé. Improving the first-miss computation in set-associative instruction caches. In *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pages 341–350, Prague, Czech Republic, July 2008.

[3] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 37–44, Delft, The Netherlands, June 2001.

[4] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for real-life processor. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, volume 2211, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[6] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE, vol.9, n7*, 2003.

[7] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET estimation. *Real-Time Systems Journal*, 34(3), Nov. 2006.

[8] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium*, pages 12–21, 1999.

[9] F. Mueller. Static cache simulation and its applications. PhD thesis, 1994.

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

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

[12] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 201–206, New York, NY, USA, 2003. ACM.

[13] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, Dresden, Germany, July 2006.

[14] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems Journal*, 1(2):159–176, 1989.

[15] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *Proceedings of IEEE Real-Time Systems Symposium*, volume 13, pages 67–91, 1997.

[16] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 148–157, Washington, DC, USA, 2005. IEEE Computer Society.

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

[18] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium*, Cancun, Mexico, 2003.

[19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.