HAL
open science

WCET ANALYSIS OF MULTI-LEVEL SET-ASSOCIATIVE DATA CACHES

Benjamin Lesage, Damien Hardy, Isabelle Puaut

To cite this version:

Benjamin Lesage, Damien Hardy, Isabelle Puaut. WCET ANALYSIS OF MULTI-LEVEL SET-ASSOCIATIVE DATA CACHES. 9th Intl. Workshop on Worst-Case Execution Time WCET Analysis, Jun 2009, Dublin, Ireland. pp.2283. inria-00531218

HAL Id: inria-00531218

https://inria.hal.science/inria-00531218

Submitted on 2 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

WCET ANALYSIS OF MULTI-LEVEL SET-ASSOCIATIVE DATA CACHES

Benjamin Lesage , Damien Hardy and Isabelle Puaut[1]

Abstract
Nowadays, the presence of cache hierarchies tends to be a common trend in processor architectures, even in hardware for real-time embedded systems. Caches are used to fill the gap between the processor and the main memory, reducing access times based on spatial and temporal locality properties of tasks. Cache hierarchies are going even further however at the price of increased complexity. In this paper, we present a safe static data cache analysis method for hierarchies of non-inclusive caches. Using this method, we show that considering the cache hierarchy in the context of data caches allows tighter estimates of the worst case execution time than when considering only the first cache level. We also present considerations about the update policy for data caches.

1. Introduction

It is crucial in hard real-time systems to prove that tasks meet their deadlines in all situations, including the worst-case. This proof needs an estimation of the worst-case execution time (WCET) of every task taken in isolation. WCET estimates have to be safe, i.e. larger than or equal to any possible execution time. Moreover, they have to be tight, i.e. as close as possible to the actual WCET. Thereof, WCET estimation techniques have to account for all possible execution paths in the program and determine the longest one (*high-level* analysis). They also have to account for the hardware the task is running on (*low-level* analysis).

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. Architectures with caches are now commonly used in embedded real-time systems due to the increasing demand for computing power of many embedded applications. The presence of caches in real-time systems makes WCET estimation difficult due to the dynamic behaviour of caches. Safely estimating WCET on architectures with caches requires a knowledge of all possible cache contents at every program point, and requires some knowledge of the cache replacement policy and, in case of data caches, update 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 [13, 14, 18, 3] have been designed and recently, extended to hierarchies of non-inclusive caches [7]. To overcome predictability issues, as the ones due to the replacement policies, is the family of approaches like locking [15, 20]. The latter methods family suits well to data caches [19, 11], whose analysis suffers from imprecise static accessed data address prediction. Indeed, if precise address prediction in the context of instruction caches has been mastered, for data caches, it remains an important concern. Nonetheless, existing methods for instruction caches have also been modified [16, 17, 4] to tackle with accesses whose target can only be over-approximated using a range of addresses.

To the best of our knowledge, no safe static cache analysis method has been proposed so far to predict worst-case data cache behaviour in the presence of a data caches *hierarchy*. The issues to be tackled when designing such an analysis are twofold. On the one hand, the prediction of cache levels impacted

[1]IRISA, University of Rennes 1, Rennes, France

by a memory reference is required to estimate the induced caches accesses. On the other hand, writes to the caches have to be considered, because they may introduce additional cache accesses. Ensued predictability problems may even get exacerbated by the lack of precise knowledge about accessed addresses.

The contribution of this paper is the proposal of a new safe static cache analysis method for multi-level non-inclusive set-associative data caches. All levels of cache are analysed sequentially. Similarly to our previous work for instruction caches [7], the safety of the proposed method relies on the introduced concept of a *cache access classification*, defining which references may occur at every cache level and have to be considered by the cache analysis of that level, in conjunction with the more traditional *cache hit/miss classification*. This paper presents experimental results showing that in most cases WCET estimates are tighter when considering the cache hierarchy than when considering the L1 cache only.

The rest of the paper is organized as follows. Related work is surveyed in Section 2.. Section 3. presents the type of caches to which our analysis applies. Section 4. then details our proposal. Experimental results are given in Section 5.. Finally, Section 6. concludes with a summary of the contributions of this paper, and gives directions for future work.

2. Related work

Caches in real-time systems raise timing predictability issues due to their dynamic behaviour 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 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* [9] or by a more compact representation called *abstract cache states* (ACS) [18, 3, 14, 13].

Two main classes of approaches [18, 13] exist for the static WCET analysis on architectures with a single level of instruction cache. In [18] the approach is based on *abstract interpretation* [2] and uses ACSs. 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 hit/miss classification* (e.g. *always hit*, *first miss*...) can then be assigned to every instruction based on the result of the three analyses. This approach originally designed for set-associative instruction caches implementing the *least recently used* (LRU) replacement policy has been extended for different cache replacement policies in [8] for instruction caches. In [13], *static cache simulation* is used to determine every possible content of the instruction cache before each instruction. Static cache simulation computes abstract cache states using data-flow analysis. A *cache hit/miss classification* is used to classify the worst-case behaviour of the cache for a given instruction. The base approach, initially designed for direct-mapped caches, was later extended to set-associative instruction caches in [14].

A peculiarity of data caches, compared to instruction caches, arises as the precise target of some references may not be statically computable. A first solution is to consider these imprecise accesses as in [17] and [4], improvements to [18]. Therefore, we base our cache analysis on these studies.

An alternative solution to deal with data caches are *Cache Miss Equations* (CME) [19, 11, 21]. To estimate cache behaviour, the iteration space of loop nests is represented as a polyhedron. *Reuse vectors* [22] are then defined between points of the iteration space. CME are set up and resolved to accurately locate misses. This method has been successfully applied to data caches in combination

with locking [19]. However, according to the authors, this approach suffers from a lack of support for data dependent conditionals.

Another scarcely addressed characteristic of data caches is the impact of memory modifying instructions. In [5], an analysis was proposed based on the *write-back* update policy. Modified data in a cache are copied back to the main memory upon eviction. The objective is thus to estimate the *write backs*, i.e. the moment when a modified data might be replaced in the cache. In a first step towards a more generic solution, we chose to use the *write-through* update policy as it removes the need of *write backs* monitoring.

Finally, about the hierarchy of data caches, we already explored a solution to this problem in [7] in the context of instruction caches. Introducing the concept of *cache access classification* (CAC), we safely identify references that may, must or never occur at every level in the cache hierarchy. This paper presents to which extent this approach can be applied to data caches.

3. Assumptions and notations

As this study focuses on data caches, code is assumed not to interfere with data in the different considered caches. There is no assumption on the means used to achieve this separation, whether they are software or hardware based. An architecture without timing anomalies, caused by interactions between caches and pipelines [12], is however assumed.

The considered cache hierarchy is composed of N levels of data caches. Each cache implements the LRU replacement policy. Using this policy, cache blocks in a cache set are logically ordered according to their age. If a cache set is full, upon a load in this set, the evicted block is the oldest one whereas the most recently accessed block is the youngest one.

A datum accessed by an instruction should be located in a single memory block. Cache line size of level L is assumed to be a multiple of the cache line size of level $L-1$. However, no assumption is made concerning the cache sizes or associativities. Furthermore, the following properties are assumed to hold:

P1.[**load**] 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.[**load**] Every time a cache miss occurs at cache level L, the entire cache line containing the missing piece of information is loaded into the cache of level L.

P3.[**store**] The modification issued by a store instruction goes all the way through the memory hierarchy. Writes to the cache levels where the written memory block is already present are triggered, along with the update of the main memory. Otherwise, if the information is absent from a cache, this cache level is left unchanged.

P4.[**store**] Upon a write in a cache block, wherever is the cache in the hierarchy, no block age modification is induced[2].

P5. There are no action on the cache contents (i.e. lookup/modification) other than the ones mentioned above.

Property P1 rules out architectures where cache levels are accessed in parallel to speed up information lookup. P2 excludes architectures with exclusive caches, whereas P5 filters out cache hierarchies ensuring inclusion.

Properties P3 and P4 on the other hand address the store instructions behaviour. P3 ensures the use of

[2]Note that this is not a strong assumption but its alleviation is left as future work.

the *write-through* update policy. In combination with P3, P4 corresponds among other things to the *write-no-allocate* update policy for members of the cache hierarchy, as illustrated in Figure 1.

Finally, the latencies to access the different levels of the memory hierarchy are assumed to be bounded and known.

We define a *memory reference* as a reference to *data* in the memory triggered by a load or store instruction in a fixed call context; a memory reference is tied to a unique instruction and a unique call context.

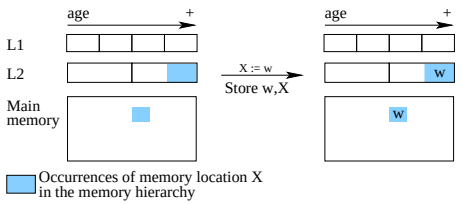


Figure 1: Example of the memory hierarchy behaviour upon a store instruction (write-through and write-no-allocate policies).

4. Data Cache Analysis

This section introduces most of the involved steps in our computation of WCET contribution for hierarchies of data caches. The structure of the whole method is outlined in Figure 2.

After a first step that extracts a *Control Flow Graph* from the analysed executable, a data address analysis (§ 4.1.) is performed. The objective is to attach to every memory reference a safe estimate of the accessed addresses.

Then, the caches of the hierarchy are analysed one after the other (§ 4.2.) based on address information. For each cache level and each instruction issuing memory operations, a *cache hit/miss classification* (CHMC) is computed. These classifications represent the worst-case behaviour of this cache level with regards to this instruction.

To be safe, the analysis of a cache level further relies on *cache access classifications* (CAC, introduced in § 4.3.) which discloses, given a memory reference, a safe approximation of whether or not it occurs at a cache level.

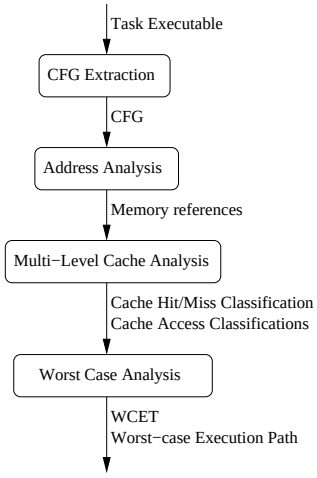


Figure 2: Complete task analysis overview

In the end, a timing analysis of memory references (§ 4.4.), considering data caches, is performed with the help of both the CHMC and CAC for each cache level. Such information may then be used to compute the longest possible execution path and finally the WCET of the task.

4.1. Address analysis

The address analysis, in the context of data caches, computes for every memory reference, the memory location it may access. Such information may however not be precisely computable, hence producing an over-approximation to yield safe values for subsequent analyses. These over-approximations take the form of ranges of possibly accessed memory blocks instead of a reference to a precise memory block.

The address analysis uses data-flow analyses which first computes stack frames addresses for each valid call context and then analyse register contents for each basic block. Considering both global and on stack accesses, the precise address of a scalar is yielded whereas the whole array address range is returned for accesses to array elements. Note that the analysis used below is the one proposed in [6].

4.2. Single level data cache analysis

Below, a data cache analysis abstracted from the multi-level aspect is introduced. All references are analysed and access filtering, according to other cache levels, is introduced later (§ 4.3.). The analysis method is presented along with the different mechanisms to handle the specificities of data caches, compared to instruction caches.

Focusing on data caches, the timing analysis of memory references is performed using the worst-case behaviour of this data cache for each memory reference. This knowledge is itself based on the cache contents at the studied program point, thus requiring a safe estimate of memory blocks present in the cache.

A *cache hit/miss classification* (CHMC) is used to model the defined cache behaviours with regards to a given memory reference. To such a reference has already been attached a set of possibly accessed memory blocks by the address analysis. This set of memory blocks is used to compute the CHMC:

always-hit (AH) : all the possibly accessed memory blocks are guaranteed to be in the cache;
first-miss (FM) : for every possibly accessed memory block, once it has been first loaded in the cache, it is guaranteed to stay in the cache afterwards;
always-miss (AM) : all the possibly accessed memory blocks are known to be absent from the cache;
content-independent (CI) : if the behaviour of the memory reference does not depend on the cache contents. The CI classification is used for store instructions, which, according to our hypotheses (§ 3.) does not depend on the cache contents;
not-classified (NC) : if none of the above applies.

Abstract cache states (ACSs) are used to collect information along the task CFG. This abstraction allows the modelling of a combination of concrete cache states, in terms of present memory blocks and their relative age. This is required as all paths have to be considered altogether to yield safe values. Different analyses, similarly to [18], are defined to collect information about cache contents at every program point. For each analysis, fixpoint computation is applied on the program CFG, for every call context:

a *Must* analysis determines if a memory block is always present in the cache, at a given point, thus allowing a *always-hit* classification;
a *Persistence* analysis determines if a memory block will not be evicted from the cache once it has been first loaded, as in the definition of the *first-miss* classification;
a *May* analysis determines if a memory block may be in the cache at a given point, otherwise the block is guaranteed not to be in the cache thus possibly allowing a *always-miss* classification. If, at a given point, some possibly accessed memory blocks are present in the May analysis but neither in the Must nor the Persistence one, the *not-classified* classification is assumed for this memory reference.

Then, for each memory reference, its set of possibly accessed memory block is compared to the memory blocks inside the input ACS computed by each analysis.

Join functions : For instructions on branch reconvergence, the $Join_{Must}$, $Join_{Persistence}$ and $Join_{May}$ (respectively for the Must, Persistence and May analysis) are used as a mean to compute input ACS:

$Join_{Must}$ computes the intersection of memory blocks present in the input ACSs, keeping for each one its maximal age as shown in Figure 3a;
$Join_{Persistence}$ keeps the union of memory blocks present in the input ACSs, as for $Join_{Must}$, the maximal age of memory blocks is kept;

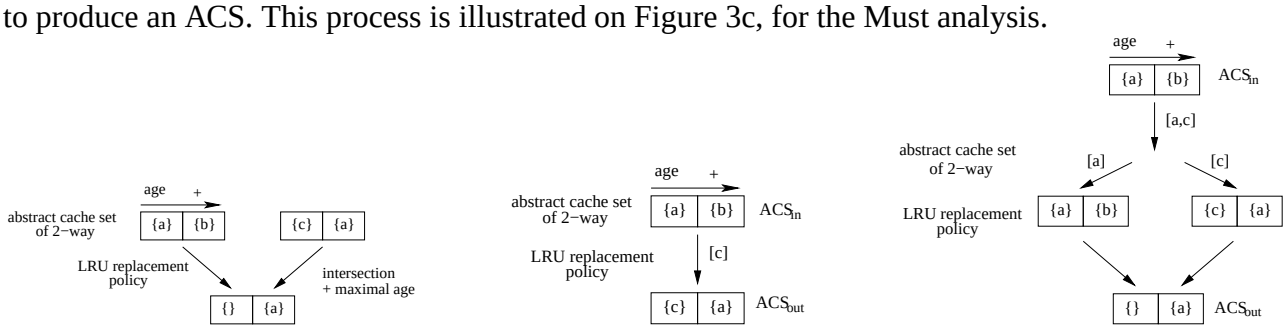
Join$_{May}$ computes the union of memory blocks present in the input ACSs, keeping for each one its minimal age.

Considering data caches, these functions are the same as the ones defined for instruction caches [18].

Update functions : The effects of memory references on the cache are modelled using the *Update*$_{Must}$, *Update*$_{Persistence}$ and *Update*$_{May}$ for the Must, Persistence and May analysis respectively. In all the cases, only load instructions have an impact on the cache contents. Store instructions have no impact on ACSs (§ 3., use of the write-through and write-no-allocate update policies). Considering data caches, the *Update* function of the different analyses is of particular interest. Indeed, it has to deal with accesses indeterminism, when a precise memory location cannot be statically defined for a load instruction. Two options have to be considered.

On the one hand, the precise accessed memory block may have been computed by the address analysis. The *Update* function is then pretty straightforward, as illustrated in Figure 3b for the Must analysis. Thus considering the *Update*$_{Must}$, the accessed block is put at the head of its cache set and the younger lines are shifted, i.e. made older and evicted if too old. Note that memory blocks outside an ACS are supposed to be older than the ones present in this same ACS. Furthermore, the different *Update* functions behaviour in this case is the same than for instructions [18].

On the other hand, when the address analysis yields a set of possibly accessed memory blocks for a memory reference, only one member of this set is actually accessed. To model this behaviour, a copy of the input ACS is created, using the appropriate *Update* function, for each possibly accessed memory block. Then, all the updated copies are unified, this time using the appropriate *Join* function to produce an ACS. This process is illustrated on Figure 3c, for the Must analysis.



(a) Join function of Must analysis

(b) Update function of Must analysis while accessing memory block c.

(c) Update function while accessing interval of blocks [a,c]

Figure 3: *Update*$_{Must}$ and *Join*$_{Must}$ functions

Termination of the analyses. In the context of abstract interpretation, to prove the termination of an analysis, it is sufficient to prove the use of a finite abstract domain and the monotony of the transfer functions. ACS domain was shown to be finite in [18]. Moreover, [18] demonstrates the monotony of the *Join*$_x$ and *Update*$_x$ functions ($x \in \{Must, Persistence$ or $May\}$), for instructions and thus for accesses to precise memory blocks. In our case, the modification to be proved is the one applied to this same *Update*$_x$ function to handle accesses to possibly referenced memory blocks.

When an ACS is updated using a set of possibly referenced memory blocks for an analysis x, a composition of the *Update*$_x$, for each block, and the *Join*$_x$ function is performed. As the composition of monotonic functions is monotonic, our modifications ensure monotony. □

4.3. Multi-level analysis

The cache analysis described in § 4.2. does not support hierarchies of caches, i.e. all references are considered for the caches by the analysis. But, a memory reference may not occur in all the

cache levels of the hierarchy. This filtering by previous caches in the hierarchy, impacts the cache contents. Hence, caches are analysed one after the other, from cache level 1 to N, and *cache access classifications* (CACs), as defined in [7], are used to represent occurrences of memory reference r on cache level L. This is possible as, according to our hypotheses, occurrences of memory references on cache level L (and thus L contents) only depend on occurrences of memory references on caches levels $L' < L$.

Different classifications have been defined to represent if memory reference r is performed on cache level L:

Always (A) means that the access *r* is always performed at cache level *L*,

Never (N) means that the access *r* is never performed at cache level *L*,

Uncertain-Never (U-N) indicates that no guarantee can be given considering the first access to each possibly referenced memory block for r, but next accesses are never performed at level L,

Uncertain (U) indicates that no guarantee can be given about the fact that the access to *r* will or will not be performed at level *L*.

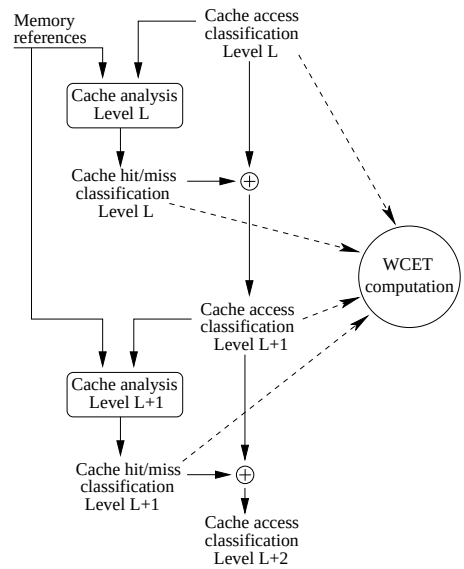


Figure 4: Multi-level non-inclusive data cache analysis framework

For the L1 cache, CAC determination is simple as it was earlier assumed that all memory references will be performed in this cache level (P1. § 3.). This implies a A classification for every memory reference, considering the L1 cache.

Concerning greater cache levels, the CAC for memory reference r and cache level L is defined using both the CAC and the CHMC of the previous level cache level (see Figure 4) as in [7], to safely model cache filtering in the cache hierarchy. This is illustrated in Table 1.

Once these classifications have been settled, they have to be considered in the multi-level cache analysis. The modifications to handle hierarchies of data caches are similar to the modifications to handle hierarchies of instruction caches [7]. A (respectively N) classification for a given instruction, means that the reference is (respectively not) performed on the considered cache level which should be modelled by the analysis. As for the U and U-N classifications, both the cases are possible: the access is performed (A) or not (N) on this cache level. Similarly to non deterministic accesses, the two alternatives are considered and their outcomes unified using the appropriate *Join* function of the analysis.

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

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

The *Update*$_x$ function, $x \in \{Must, Persistence$ or $May\}$, is modified to consider these classifications :

$$Update_x^{ml}(ACS_{in}, r, L) = \begin{cases} Update_x(ACS_{in}, r, L) & \text{if } CAC_{r,L} = A \\ ACS_{in} & \text{if } CAC_{r,L} = N \\ Join_x(ACS_{in}, Update_x(ACS_{in}, r, L)) & \text{if } CAC_{r,L} = U \vee CAC_{r,L} = \textit{U-N} \end{cases}$$

where *Update*$_x^{ml}$ represents the multi-level version of the *Update*$_x$ functions presented earlier (§ 4.2.). This definition can be inflected to *Update*$_{Must}^{ml}$, *Update*$_{Persistence}^{ml}$ or *Update*$_{May}^{ml}$ using the corresponding *Update* and *Join* functions of the Must, Persistence or May analysis respectively.

Termination of the analyses. The differences between the multi-level data cache analysis and the single-level data cache analysis are the *Update*$_x$ functions, $x \in \{Must, Persistence$ or $May\}$, for the different analyses. However these functions were proved to be monotonic in section 4.2.. The proof in [7] thus holds for data caches as well. We need to demonstrate that the *Update*$_x^{ml}$ function is monotonic for the four possible values of CAC.

For an A access, *Update*$_x$ and *Update*$_x^{ml}$ behave identically. *Update*$_x$ being monotonic, *Update*$_x^{ml}$ is also monotonic. Considering a N access, *Update*$_x^{ml}$ is the identity function and so is monotonic. Finally, considering an U or U-N access, *Update*$_x^{ml}$ composes *Update*$_x$ and *Join*$_x$. These two functions are monotonic, so is their composition. Thus *Update*$_x^{ml}$ is monotonic which guarantees the termination of our analysis. □

4.4. WCET computation

CHMCs represent the worst-case behaviour of the cache given a memory reference. They are useful to compute the contribution of references to the WCET. This contribution can then be used in existing methods to compute the WCET. In our case, we focus on IPET based methods which estimates the WCET by solving an *Integer Linear Programming* (ILP) problem [10].

The timing of the memory reference r, with regards to the data caches, is divided in two parts. $first$ and $next$ respectively distinguish the first and successive iterations of loops. We define $COST_first(r)$ and $COST_next(r)$ as the respective contribution to the WCET of memory reference r for the first and successive iterations of the loop in which the reference is enclosed, if any[3]. If no loop encloses r, $COST_first(r)$ will be implicitly preferred, by the ILP solver, over $COST_next(r)$ as the cost of r. Indeed $COST_first(r)$ accounts for all first misses of memory reference r and $COST_first(r) \geq COST_next(r)$.

With $freq_r$ a variable computed in the context of IPET analysis and representing the execution frequency of r along the worst-case execution path of the task, the following constraints are defined: $freq_r = freq_{first,r} + freq_{next,r}$ and $freq_{first,r} \leq 1$. The WCET contribution of the reference r with regards to data caches is then defined as:

$$WCET_data_contribution(r) = COST_first(r) \times freq_{first,r} + COST_next(r) \times freq_{next,r}$$

As we focus on an architecture without timing anomalies, it is safe to consider that NC references behave like AM ones, which are the worst-case on our architecture. Therefore, U accesses to a cache level behave as A ones, from the timing analysis considering data caches point of view[4]. We define $always_contribute(r)$ and $never_contribute(r)$ as the sets of memory hierarchy levels which respectively always or never contribute to the execution latency of memory reference r, with $M = N + 1$ the main memory in the memory hierarchy:

$$never_contribute(r) = \{L \mid 1 \leq L \leq M \wedge CAC_{r,L} = N\}$$

$$always_contribute(r) = \{L \mid 1 \leq L \leq M \wedge (CAC_{r,L} = A \vee CAC_{r,L} = U)\}$$

One option to deal with a FM classification for reference r on cache level L would be to consider, for every execution of r, that cache level $L + 1$ is accessed: $CAC_{r,L} =$ U-N $\Rightarrow L \in always_contribute(r)$. However, it might be overly pessimistic with regards to the semantic of the FM and the de facto inherited U-N classifications. We need additional notations to depict this behaviour and tighten r contribution to the WCET.

Given a memory reference r and a cache level L, let $memory_blocks_{r,L}$ be the set of r target memory blocks on cache level L, as computed by the address analysis. This information is computed

[3]Remember that r is contextual and attached to a unique instruction. A memory reference tied to the same instruction, but another context may be enclosed in different loops.

[4]From the ACS computing point of view, they keep different meanings.

using the cache blocks size of cache level L. We only need to know a bound on the size of this set:$|memory_blocks_{r,L}| = \lceil \frac{Addr_range_r}{cache_block_size_L} \rceil$ with $Addr_range_r$ the size of the address range computed by the address analysis for memory reference r and $cache_block_size_L$ the size of level L cache blocks.

Furthermore, we define max_freq_r as the maximum statically computable execution frequency of reference r. Such an information is computed as the product of the maximum number of iterations of all loops containing r: $max_freq_r = \prod_{lo \in Loops(r)} max_iter_{lo}$, where $Loops(r)$ are the loop containing the memory reference r and max_iter_{lo}, the maximum iteration attribute for loop lo. Note that the whole task is itself considered to be a loop enclosing all memory references and whose $max_iter = 1$, ensuring that the max_freq attribute of an otherwise not enclosed in a loop memory reference is 1.

Each element of $memory_block_{r,L}$ produces at most one miss in the cache level L, the first time it is accessed, according to the definition of the FM classification. Thus, reference r will not produce more than $min(max_freq_r, |memory_blocks_{r,L}|)$ misses on cache level L.

Note that $min(freq_r, |memory_blocks_{r,L}|)$, with $freq_r$ the execution frequency of r on the worst-case execution path of the task, would be a tighter bound. However, this bound is required to compute $freq_r$ and vice versa thus leading to a chicken-and-egg problem.

Once these elements have been defined, we can bound the number of occurrences of memory reference r on cache level L:

$$max_occurrence(r, L) = \begin{cases} 0 & if\ L \in never_contribute(r) \\ max_freq_r & if\ L \in always_contribute(r) \\ min(|memory_blocks_{r,L-1}|, max_occurrence(r, L-1)) & if\ CHMC_{r,L-1} = FM \\ max_occurrence(r, L-1) & otherwise \end{cases}$$

Intuitively, if cache level L is never (respectively always) accessed by memory reference r, there cannot be any (respectively more than max_freq_r) occurrences of r on cache level L. Similarly, there cannot be more occurrences of r on cache level L than on cache level $L-1$. Finally, if $CHMC_{r,L-1} = FM$, only the first accesses to memory blocks belonging to $memory_blocks_{r,L-1}$ will occur on cache level L.

The definition of $COST_next(r)$ is pretty straightforward, as we only have to count the access latency of cache levels L which are guaranteed to always contribute to the timing of r. Other accesses are considered in $COST_first(r)$:

$$COST_next(r) = \begin{cases} \sum_{L \in always_contribute(r)} ACCESS_latency_L & \text{if r is a load of data} \\ STORE_latency & \text{if r is a store of data} \\ 0 & \text{otherwise} \end{cases}$$

$COST_first(r)$ is a bit harder to define. In some cases, we have a bound on the number of occurrences of a memory reference r on cache L. As previously stated, the solution of not considering these bounds might be unnecessary pessimistic. Therefore, we chose to use the $COST_first(r)$ to hold these additional latencies. It could be understood as considering all the possible first misses in the first execution of r, in addition to all the always accessed cache levels latencies:

$$COST_first(r) = \begin{cases} \sum_{L \in always_contribute(r)} ACCESS_latency_L + \sum_{CAC_{r,L}=\text{U-N}} ACCESS_latency_L \times max_occurrence(r, L) & \text{if r loads data} \\ STORE_latency & \text{if r stores data} \\ 0 & \text{otherwise} \end{cases}$$

5. Experimental results

5.1. 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 and with the default linker memory layout. The WCETs of tasks are computed by the Heptane timing analyser [1], more precisely its Implicit Path Enumeration Technique (IPET). The timing of memory references with regards to data caches is evaluated using the method introduced in Section 4.4.. The analysis is context sensitive (functions are analysed in each different calling context). To separate the effects of caches from those of the other parts of the processor micro-architecture, WCET estimation only takes into account the contribution of caches to the WCET. The effects of other architectural features are not considered. In particular, timing anomalies caused by interactions between caches and pipelines, as defined in [12] are disregarded. The cache classification *not-classified* is thus assumed to have the same worst-case behaviour as *always-miss* during the WCET computation in our experiments. The cache analysis starts with an empty cache state.

Name	**Description**	**Code size** (bytes)	**Data size** (bytes)	**Bss size** (bytes)	**Stack size** (bytes)
crc	Cyclic redundancy check computation	1432	272	770	72
fft	Fast Fourier Transform	3536	112	128	288
insertsort	Insertsort of an array of 11 integers	472	0	44	16
jfdctint	Fast Discrete Cosine Transform	3040	0	512	104
ludcmp	Simultaneous Linear Equations by LU Decomposition	2868	16	20800	920
matmult	Product of two 20x20 integer matrixes	1048	0	4804	96
minver	Inversion of floating point 3x3 matrix	4408	152	520	128
ns	Search in a multi-dimensional array	600	5000	0	48
qurt	Root computation of quadratic equations	1928	72	60	152
sqrt	Square root function implemented by Taylor series	544	24	0	88
statemate	Automatically generated code by STARC (STAtechart Real-time-Code generator)	8900	32	290	88

Table 2: Benchmark characteristics

Benchmarks. The experiments were conducted on a subset of the benchmarks maintained by Mälardalen WCET research group[5]. Table 2 summarizes the characteristics characteristics (size in bytes of sections *text*, *data*, *bss* (unitialized data) and *stack*).

Cache hierarchy. The results are obtained on a 2-level cache hierarchy composed of a 4-way L1 data cache of 1KB with a cache block size of 32B and a 8-way L2 data cache of 4KB with a cache block size of 32B. A perfect private instruction cache, with an access latency of one cycle, is assumed. All caches are implementing a LRU replacement policy. Latencies of 1 cycle (respectively 10 and 100 cycles) are assumed for the L1 cache (respectively the L2 cache and the main memory). Upon store instructions a latency of 150 cycles is assumed to update the whole memory hierarchy.

5.2. Considering the cache hierarchy

To evaluate the interest of considering the whole cache hierarchy, two configurations are compared in Table 3. In the first configuration, only the L1 cache is considered and all accesses to the L2 cache are defined as misses ($WCET_{L1}$, column 1). In the other configuration ($WCET_{L1\&L2}$, column 2), both the L1 and the L2 caches are analysed using our multi-level static cache analysis (§ 4.3.). Table 3 presents these results for the two cases using the tasks WCET as computed by our tool and based on our multi-level static analysis. The presented WCET are expressed in cycles.

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

The third column of Table 3 represents the improvement, for each analysed task, attributable to the consideration of the second cache level of the hierarchy: $\frac{WCET_{L1}-WCET_{L1\&L2}}{WCET_{L1}}$.

Benchmark	**WCET considering only a L1 cache** (cycles)	**WCET considering both L1 and L2 cache** (cycles)	**Improvement considering the L2 cache**
crc	9373240	9119240	2.7 %
fft	14068000	6320470	55.07 %
insertsort	456911	456911	0 %
jfdctint	597239	464239	22.26 %
ludcmp	1778460	1778460	0 %
matmult	24446200	24446200	0 %
minver	328506	303626	7.57 %
ns	861575	861575	0 %
qurt	622711	520531	16.4 %
sqrt	94509	94509	0 %
statemate	1303760	1129380	13.37 %

Table 3: Evaluation of the static multi-level n-way analysis (4-way L1 cache, 8-way L2 cache, cache sizes of 1KB (resp. 4KB) for L1 (resp. L2)).

Considering the L2 cache using our multi-level static analysis, may lead up to 55.07% improvement for the *fft* task. In average, the results are still interesting 10.67% (6.23% whithout *fft*) with the exceptions of a subset of tasks not taking any benefit in the consideration of the L2 cache (*insertsort*, *ludcmp*, *matmult*, *ns* and *sqrt*).

The issues raised by these tasks are twofold. On the one hand, tasks like *sqrt* and *insertsort* use little data. This small working set fits in the L1 cache. Thus, the L2 cache is only accessed upon the very first misses. On the other hand are tasks like *ludcmp*, *matmult* and *ns* accessing large amount of data that fit neither in the L1 nor the L2 cache. *ludcmp* accesses a small portion of a large array of floats, which probably shows temporal locality but, as our address analysis returns the whole array range, this precision is lost. Concerning *matmult* and *ns*, they consist of loop nests running through big arrays. Again, there is temporal locality between the different iterations of the most nested loop. However, this locality is not captured as the FM classification applies to the outermost loop in such cases, and thus consider the whole array as being persistent or not.

Other tasks exhibit such small data set but, due to the analysis pessimism, some memory blocks might be considered as shifted outside the L1. However, they are detected as persistent in the L2 (*fft*, *jfdctint*, *minver*, *qurt*, *statemate* and to a lesser extent *crc*).

6. Conclusion

In this paper we presented a multi-level data cache analysis. This approach considers LRU set-associative non-inclusive caches implementing the *write-through*, *write-no-allocate* update policies. Results shows that considering the whole cache hierarchy is interesting with a computed WCET contribution of data caches being in average 10.67% smaller than the contribution considering only the first level of cache. Furthermore, the computation time is fairly reasonable, results for the eleven presented benchmarks being computed in less than a couple of minutes.

Keeping the same study context, future researches include the definition of less pessimistic constraints on the number of misses or improving the precision of the different analyses to handle tighter classifications (e.g. a classification per accessed block instead of per memory reference). Extending the analysis to other data cache configurations, whether speaking of replacement policies (e.g. pseudo-LRU) or update policies (e.g. write-back or write-allocate) might be the subject of other studies. Finally, the extension of the analysis context to handle multi-tasking systems or multi-core architectures is left as future works.

References

[1] 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.

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

[3] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard 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, October 2001.

[4] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30. Springer-Verlag, 1998.

[5] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, 1999.

[6] Damien Hardy and Isabelle Puaut. Predictable code and data paging for real time systems. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 266–275. IEEE Computer Society, 2008.

[7] Damien Hardy and Isabelle Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 456–466. IEEE Computer Society, 2008.

[8] 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.

[9] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[10] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 456–461. ACM, 1995.

[11] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, page 255. IEEE Computer Society, 1999.

[12] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12. IEEE Computer Society, 1999.

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

[14] Frank Mueller. Timing analysis for instruction caches. 18(2-3):217–247, 2000.

[15] 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.

[16] Harini Ramaprasad and Frank 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. IEEE Computer Society, 2005.

[17] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 203–212. ACM, 2007.

[18] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. 18(2-3):157–179, 2000.

[19] Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium*, Cancun, Mexico, 2003.

[20] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *Trans. on Embedded Computing Sys.*, 7(1):1–38, 2007.

[21] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 175. IEEE Computer Society, 2002.

[22] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44. ACM, 1991.