A Generic Framework to Integrate Data Caches in the WCET Analysis of Real-time Systems[*][†]

Juan Segarra, Rubén Gran Tejero, Víctor Viñals
Dpt. Informática e Ingeniería de Sistemas, I3A, Universidad de Zaragoza, 50018 Spain
{jsegarra,rgran,victor}@unizar.es

December 23, 2021

Abstract

Worst-case execution time (WCET) analysis of systems with data caches is one of the key challenges in real-time systems. Caches exploit the inherent reuse properties of programs by temporarily storing certain memory contents near the processor, in order that further accesses to such contents do not require costly memory transfers. Current worst-case data cache analysis methods focus on specific cache organizations (set-associative LRU, locked, ACDC, etc.), most of the times adapting techniques designed to analyze instruction caches. On the other hand, there are methodologies to analyze the data reuse of a program, independently of the data cache. In this paper we propose a generic WCET analysis framework to analyze data caches taking profit of such reuse information. It includes the categorization of data references and their integration in an IPET model. We apply it to a conventional LRU cache, an ACDC, and other baseline systems, and compare them using the TACLeBench benchmark suite. Our results show that persistence-based LRU analyses dismiss essential information on data, and a reuse-based analysis improves the WCET bound around 17% in average. In general, the best WCET estimations are obtained with with optimization level 2, where the ACDC cache performs 39% better than a set-associative LRU.

[*]This work was supported by MINECO/AEI/ERDF (EU) (grant PID2019-105660RB-C21), and Aragón Government (T58_20R research group).

[†]The use, investigation or development, in a direct or indirect way, of any of the scientific contributions of the authors contained in this work by any army or armed group in the world, for military purposes and for any other use which is against human rights or the environment, is strictly prohibited unless written consent is obtained from all the authors of this work, or all the people in the world.

1 Introduction

Real-time systems are increasingly present in industry and daily life. We can find examples in many sectors including avionics, robotics, automotive processes, manufacturing, and air-traffic control. A real-time system consists of a number of tasks with a required functionality. These tasks have to be scheduled in a way that they meet their deadlines. To ensure that this occurs, and hence that the system operates correctly, worst-case execution time (WCET) and schedulability have to be analyzed.

Analyzing the interactions between the program and the hardware is a complex part, since current processors perform many operations with a variable duration in order to improve performance. In particular, the memory subsystem services the processor with variable latency and can be the greatest contribution to the WCET. A memory hierarchy made up of one or more cache levels exploits program reuse and saves execution time and energy consumption by delivering data and instructions with an average latency of a few processor cycles instead of requiring costly memory transfers. Although cache designs are ubiquitous in contemporary processors, many details regarding them are still ignored in the WCET analysis, and single-level LRU (Least Recently Used) instruction caches are still an open issue [32]. This situation is even worse for data caches, since writing policies must be also modeled. Most conventional data caches are writeback (store instructions write just the cached data, and memory is updated when the corresponding dirty cache line is evicted), which in general results in fewer memory transfers than writethrough policy (store instructions write to the cache as well as to the main memory) [19]. In turn, both writeback and writethrough policies can be combined with dif-

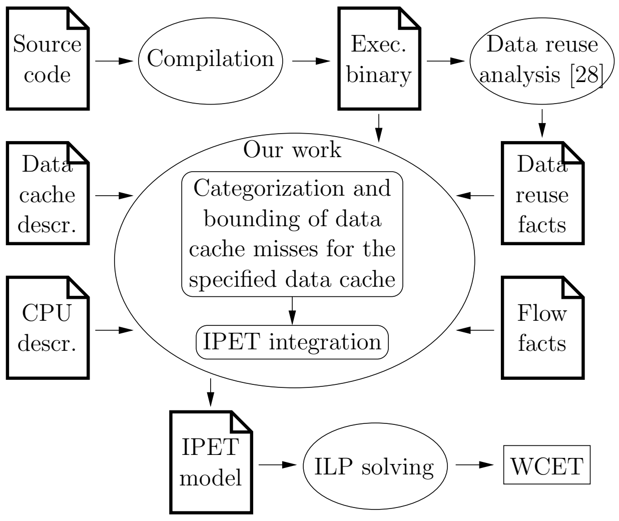


Figure 1: Context of our proposal. The compilation may be performed by any compiler, with any optimization options. The data cache description includes the cache organization (set-associative LRU, ACDC, etc.), its policies (write-back, write-through, write-allocate, fetch-on-write-miss, write-around, etc.), and its configuration (number of sets, ways, size, etc.).

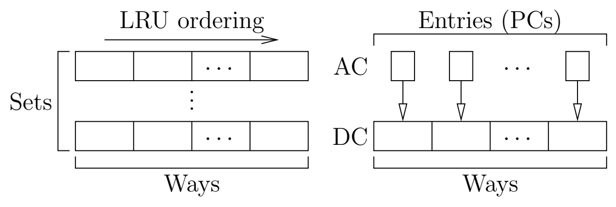
ferent write-miss allocation policies. Also, the interaction between the code and the data cache is much more complex than with the instruction cache. This can be seen in common scenarios such as loops, function calls, and execution-time address computation. In loops, a memory instruction may access different data memory addresses depending on the loop iteration. In functions, memory instructions accessing local variables use stack frames, whose base address depends, among other things, on the nesting level. Regarding address computation, a memory instruction may access a data-dependent memory address unknown at compilation/static analysis time. Such complexity to bound data cache hits and misses is hard enough so that, to the best of our knowledge, there is no comprehensive comparison regarding the impact of data caches to the WCET.

In this paper we propose a generic framework to analyze data caches (see Figure 1). Unlike most methods focused on analyzing which contents are cached at a given program point for a particular cache configuration (e.g. [17, 32]), our approach is applied on top of a reuse analysis of the compiled code [28]. We let such analysis to explore the reuse as a property of the binary code (i.e. how data are reused, independently of being cached or not), and then we analyze whether a particular cache organization is able to exploit the detected data reuse. Our proposal includes basic categories to classify data references, similar to those used for the instruction cache, and how such categories are translated to the Implicit Path Enumeration Technique (IPET) to obtain the WCET bound. We describe the implementation of a conventional set-associative LRU data cache (writeback, with write-allocate and fetch on write-miss), a predictable ACDC cache [29], an unlimited size LRU cache, and a system without cache to compare to. For the ACDC cache, we also detail a new method to configure it heuristically. Since our proposals take profit of the reuse information, our hit/miss bounds are much more precise than current state of the art approaches. Such precision allows us to perform a detailed comparison of the impact of the considered data cache organizations to the WCET. Our results show that, with our framework, the estimated WCET bound with LRU data caches is reduced 17.23% in average with respect to existing methods. Also, the WCET bound with the predictable cache ACDC is reduced another 19.62% in average in respect of a system with LRU data cache.

Our contributions can be summarized as follows:

- Generic framework for the analysis of data caches in the worst case based on reuse information, applicable to any data cache organization.
- Description of its application to a conventional LRU cache, an ACDC cache, and an LRU cache with unlimited size.
- Integration of the above WCET analysis into the IPET technique.
- Heuristic method to obtain good ACDC configurations.
- Experimental comparison of data hit ratio and WCET bound for different data cache organizations, analysis methods, and compiler optimization levels performed on ARM v7 binaries.

The rest of the paper is organized as follows. Section 2 describes related work, including a brief description of the ACDC cache. Section 3 details our proposed generic framework for the data cache WCET analysis. Its integration into IPET is described in Section 4. Then, Section 5 discusses the safety of our approach. In order to analyze the ACDC, we propose a simple method to configure it in



(a) Set-associative LRU cache.

(b) ACDC.

Figure 2: Schematics of set-associative LRU and ACDC caches. The required hardware to hold the LRU ordering of the ways in each set in the set-associative cache is not represented.

Section 6. Next, Section 7 describes our experimental environment and our results. Finally, Section 8 presents our conclusions.

2 Related work

Most current set-associative LRU cache (Figure 2a) analysis methods are based on the *must/may* analysis [13]. This method uses *Abstract Interpretation* [11] to determine the possible contents of the cache without requiring explicit address sequences. A recent improvement on this methodology achieves its maximum possible precision [32]. Although it works well on instructions, its application on data is limited due to the fact that it is based on accesses to known and constant addresses at compile time. For instance, if a given memory instruction accesses a different data memory address each time it is executed (e.g. an array traversal), the accessed addresses must be tagged as unknown. As a future work, authors suggest that, with a preprocessing analysis, it might be possible to bound the address space of a given memory instruction, e.g. to an array. If such unknown access is repeated over time so that the amount of data accessed is larger than their address space, some hits may be guaranteed. However, guaranteeing that a memory access does not go out of bounds is not trivial, and caches are not designed to hold large data structures. Up to our knowledge, feasibility of such preprocessing analysis has not been addressed yet.

Indeed, when whole data structures fit in cache, scratchpad memories or locked data caches would be preferable [34]. Locked data caches are not explored in this work, since they are completely dependent on the data size of the task and the cache size. That is, their worst-case performance would be equal or better than any other option if all data fit in cache, and worse than any other option when the percentage of cached data is below certain threshold.

Although most WCET studies on caches assume a single level hierarchy, there are a few that analyze multi-level caches [9, 21]. Essentially, they apply the must/may analysis to each cache level. Our paper does not consider multi-level caches, but a similar level-by-level approach could be applied. Regarding data cache write policies, most studies focus on writethrough (e.g. [18]), but writeback caches provide a better WCET bound [6, 36].

Alternatively to set-associative LRU caches, there are designs of predictable caches for real-time systems [15, 29]. Instead of conventional data-driven caches, the ACDC (Address-Cache Data-Cache) is a small instruction-driven data cache that effectively exploits reuse [29]. It operates from a fixed preselected subset of load/store instruction addresses held in the AC part of the ACDC cache (see Figure 2b). Such selected load/store instructions have data cache replacement permission (DRP). Each permission is associated with a particular data cache line in the DC part of the ACDC. Thus, when executing a load-/store instruction that misses in DC, the replacement of the data line on DC will be only allowed if such an instruction has DRP (i.e., the PC of this instruction is kept in an AC entry). Since each selected memory instruction replaces its own data cache line, pollution is prevented and performance is independent of the size of the data structures in tasks. Figure 3 shows the flowchart of the ACDC behavior. For data accesses, there is a fully-associative look-up, so that any access may benefit from the cached content. On miss, if the missing load/store has DRP (its PC is in AC), the DC line assigned to this load/store is replaced, as in a conventional writeback write-miss allocate cache. However, misses triggered by instructions without DRP bypass DC. That is, loads bring the specified data to the processor without modifying DC, and stores write directly to main memory without fetching the missing data, as in a write-around cache [19]. There is a similar proposal, also based on granting replacement permissions to specific memory instructions, focused on temporal reuse for large data structures [15]. However, it is not designed as a general purpose predictable cache, but as an auxiliary cache targeted to codes optimized by tiling/blocking transformations.

Independently of the target data cache, the effects of context switches to the WCET are important.

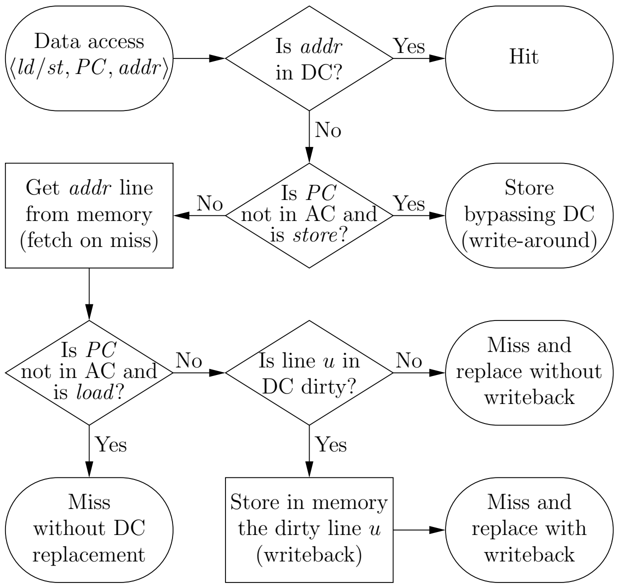

Data access
⟨ld/st, PC, addr⟩
Is addr in DC?
Yes
Hit
No
Get addr line from memory (fetch on miss)
No
Is PC not in AC and is store?
Yes
Store bypassing DC (write-around)
Is PC not in AC and is load?
No
Is line u in DC dirty?
No
Miss and replace without writeback
Yes
Yes
Miss without DC replacement
Store in memory the dirty line u (writeback)
Miss and replace with writeback

Figure 3: ACDC operation flow chart for a *load*/*store* instruction (in PC) to $addr$, which may evict a cache line u. PCs in AC are those with DC replacement permission. Notice that the first three decisions can be evaluated in parallel.

There are many proposals to bound the WCET increase related to context switches. Due to the small size of the ACDC, the best option would probably be to save and restore the cached data on every context switch, as proposed for lockable instruction caches [3]. For larger caches, preemption costs are higher, and other approaches may be preferable [20, 31]. Although adding the constant bound of preemption costs to the WCET estimation is trivial, it is not included in this study, since this cost depends very much on the tasks in the multitasking system.

As outlined above, the goal of caches is to keep local copies of data that are likely to be used again. Cache Miss Equations (CMEs) [14] are based on the reuse theory [35], and have been used to test whether it is worth caching whole data structures [34]. However CMEs are limited to perfectly nested loops. A similar approach is able to also analyze imperfectly nested loops by transforming them to perfectly nested loops with guards [8]. Although such transformation extends the analyzable loops, the analysis of other constructs, and therefore of general programs, is not possible with this approach. In order to analyze general codes, symbolic names and a congruence analysis can be used to determine whether accesses are mapped to the same LRU cache set/line [17]. In this way hit/miss information may be obtained even for accesses to unknown memory addresses. Such approach identifies certain *group reuse* cases (two different instructions accessing to the same data address), but does not perform a whole reuse analysis [35]. A more recent work provides the theoretical foundations to perform a safe reuse analysis on tasks [28]. It uses polyhedra to track the content of registers and memory by means of Abstract Interpretation [11]. Then the access patterns of data references are extracted, along with its reuse information [35]. So, it is able to analyze whole programs and generate their *data reuse facts* with equal or more precision than previous methods. However, it does not address its practical application to any specific data cache, nor its integration in a WCET analysis [28]. We propose a generic methodology for data cache analysis and its WCET analysis (see Figure 1) that exploits these *data reuse facts* similarly to the more common *flow facts*.

3 Generic data cache analysis

In order to evaluate the impact of caches in the WCET, it is required to predict their behavior in the worst case as accurately as possible. Typically, this is performed through an analysis of memory references that classifies them into a few categories, and then integrates such references into the WCET analysis based on their category. So, from now on, we call *reference* to a memory operation in the (static) binary code, and we call *access* to an actual execution (dynamic) of a memory reference. For instance, for a typical array traversal in a loop, the load instruction in the code contains a single memory reference for traversing the array, which is translated into multiple accesses to the specific address of each element in the array for each loop iteration. This section describes how we classify each data reference depending on the target data cache by means of reuse information.

3.1 Intuitive example

To illustrate the hit/miss computation on data in the worst case, let us consider the optimized matrix multiplication code in Figure 4. In this example, the four existing references (three loads and one store) result in accesses to different addresses, so an address-based analysis would consider such accessed addresses as unknown. For this code, Figure 5a shows the representation of a typical persistence analysis [32]. In the worst case, conflicts must be assumed between

```
for  (i=0;i<n;i++)
  for  (k=0;k<n;k++) {
    t=B[i][k];
    for  (j=0;j<n;j++)
      A[i][j] = A[i][j] + t*C[k][j];  }
```

Figure 4: Optimized matrix multiplication code, $A = B \times C$, assuming matrix A initialized to zero. Induction variables i, j, k, and temporal variable t should be allocated to registers.

all accesses, and with such assumption all cached data lines must be safely considered as evicted before being reused, resulting in always miss / no persistence in all references. Notice that, even considering address subspaces (in this case the rows of a matrix), interleaving accesses raise uncertainty regarding conflicts in the worst case. Moreover, if matrices are parameters of functions or they are processed by pointers (such as, for instance, *matrix1* benchmark in TACLeBench [12]), bounding the address space of loads/stores may be impossible. So, except for constant addresses (global scalar variables), address-based analyses are not adequate to predict the hits/misses on data caches.

Alternatively, Figure 5b shows a representation considering the access patterns and reuse for the code in Figure 4 [28]. It can be seen that each load/store is associated with a linear access pattern that results in accesses to sequential elements in each array (self spatial reuse), and there is group temporal reuse between the load and the store to `A[i][j]`. This information is cache-independent, so the corresponding reuse analysis must be performed just once. Then, it can be used to obtain the hits/misses in the worst case for any particular data cache. For instance, assuming an LRU cache with just 2 ways, both A and C loads will have a high hit ratio (depending on how many array elements fit in each cache line) and the store to A will always hit, since it reuses the data previously loaded. Notice that this can be asserted even if the base addresses of the matrices are unknown. Also, group reuse can be set even on unknown addresses/patterns, as long as it can be guaranteed that two references access to the same memory address. As it can be seen, an accurate hit/miss analysis cannot disregard data access patterns nor reuse information.

Table 1: Categories for data memory references.

Category	Description
Always-hit (AH)	All accesses hit
First-miss (FM)	When iterating in its enclosing loop, only the first access may miss
k-miss (KM)	When iterating in its enclosing loop, only up to k accesses may miss
First-hit (FH)	All accesses but the first one may miss
Not-classified (NC)	All accesses may miss

3.2 Limitations of this work

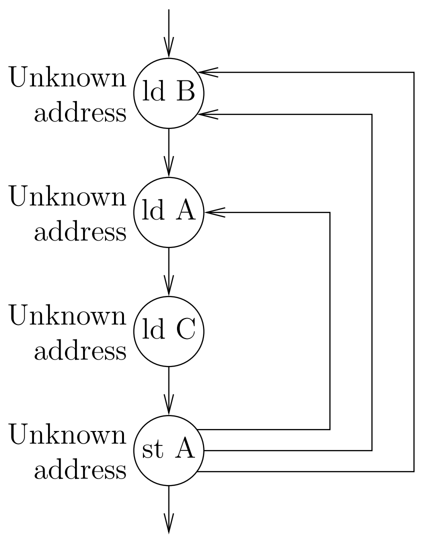
As outlined in Section 1, our framework requires a previous reuse analysis of the compiled code [28]. Currently, such analysis does not deal with recursive functions. It cannot analyze non-natural loops (loops with more than one entry point) either, which may appear in particular uses of *goto* statements.

Although the data reuse facts provided by this previous analysis for a given binary code are safe, it is important to notice that programs are sequences of instructions to be executed in order, so this safety may not hold if such an order is not respected. Therefore, processors that may not access data in program order are not analyzable with our proposal. Such behavior might be found in out-of-order execution, data prefetchers, or speculative execution of loads/stores (e.g., performing memory accesses in a mistaken branch).

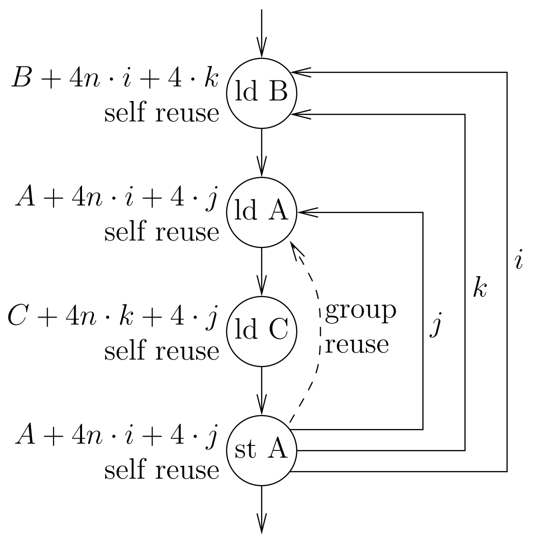
3.3 Categories of data references

In order to calculate whether a given memory access will result in a data hit or miss, most analysis methods first classify memory references into categories, and then a hit/miss computation is performed based on these categories. In this way, hits/misses are not calculated per memory access (which would be intractable), but per memory reference.

Table 1 shows our proposed categories for data references. Except for the k-miss (KM) category, explained below, they are adaptations of the typical hit/miss categories used in instruction cache analysis [13, 25, 33]. Also, Table 1 includes their description, since these categories have slightly different/ambiguous meanings in previous papers [5]. Since data accesses present much uncertainty, our cate-



(a) Persistence analysis.



$B + 4n \cdot i + 4 \cdot k$ self reuse
ld B
$A + 4n \cdot i + 4 \cdot j$ self reuse
ld A
$C + 4n \cdot k + 4 \cdot j$ self reuse
ld C
$A + 4n \cdot i + 4 \cdot j$ self reuse
st A
group reuse
j
k
i

(b) Reuse analysis.

Figure 5: Representation of different data access analyses for the code in Figure 4, with elements of 4 B.

gories bound the number of misses, but not the number of hits. This safe approach makes it possible to dispense with an overlapping data analysis. Another important detail is that first-miss (FM) and KM categories are associated with the deepest enclosing loop containing the corresponding memory instruction (e.g., in Figure 4, loop k encloses the reference to B, and loop j encloses references to A and C). For instance, a FM reference in the deepest loop of a nested loop structure means that, at most, it will miss as many times as the deepest loop is reached.

Apart from previous categories, we introduce the KM category. It has an extra parameter k that specifies the calculated constant bound on the number of misses for a given reference in its enclosing loop. Having the access pattern of a self-reusing reference, obtaining its bound k is trivial. For instance, let us take $A + 4n \cdot i + 4 \cdot j$ from Figure 5b, which sequentially traverses a matrix row of 4 byte elements in its enclosing loop (j). Let us assume cache lines of $4L$ bytes (L elements per cache line). Also, let us assume aligned $n \times n$ matrices, with n multiple of L. In such case, each row would need $4n/4L$ cache lines, that is, $k = n/L$ misses at most in loop j. This calculation may be trickier if the array base address is unaligned or unknown, and may include overestimation if part of the array is already cached.

3.4 Classification of references into categories

The classification of references into categories depends on the target data cache, since not all cache organizations exploit the data reuse in the same way. Essentially, we rely on the minimal cache life-span metric (*mls()* function in algorithms below), which determines the minimum number of accesses necessary to evict an element that has just been accessed [27]. Apart from the cache organizations detailed below, other caches can be analyzed in a similar way by means of this metric.

3.4.1 Background on data reuse theory

Access patterns (e.g., $A + 4n \cdot i + 4 \cdot j$) are based on loop nest data reuse theory [35]. For the sake of a self-contained research, let us briefly introduce such a theory, as it is outlined in previous papers [28,29]. Each iteration in a loop nest corresponds to a node in its *iteration space*. In a loop nest of depth n, this node is identified by its induction variables vector $\vec{i} = (i_1, i_2, \ldots, i_n)$, where i_j is the iteration value of the jth loop in the nest, counting from the outermost to innermost loop. Let d be the number of dimensions of an array A. The reference $A[\vec{f}(\vec{i})]$ is said to be uniformly generated if $\vec{f}(\vec{i}) = H\vec{i} + \vec{c}$, where $\vec{f}$ is an indexing function $Z^n \to Z^d$, the $d \times n$ matrix H is a linear transformation, and $\vec{c}$ is a constant vector.

Row k in H represents the linear combination of the induction variables corresponding to the kth array index. Since any data structure is mapped to memory and memory can be seen as a single dimension space, $\vec{f}$ can be transformed into an equivalent $f(\vec{i}) = \vec{h} \cdot \vec{i} + c$. So, for reference $A + 4n \cdot i + 4 \cdot j$ from Figure 5b, $\vec{i} = (i, k, j)$, $\vec{h} = (4n, 0, 4)$, and $c = A$. A given reference is constant if all elements in $\vec{h}$ are 0, and it is array otherwise. If the reference cannot be described as $\vec{h} \cdot \vec{i} + c$, then it is non-linear.

3.4.2 Conventional set-associative LRU cache

Let us begin with the classification for conventional LRU caches. In order to classify data references into categories, multiple approaches can be used. For references to constant addresses, a classification via *Binary Decision Diagrams* is probably the most efficient [32]. In addition, references with group reuse could be accurately analyzed if such information were provided [28]. However, integrating linear patterns might not be possible, due to the fact that the accessed address varies for each access. Since our goal is to evaluate the impact of an LRU data cache in the WCET bound with precision, we implement an analysis based on the reuse information (Algorithm 1).

For references *without group-reuse* (Algorithm 1, line 36), hits are possible if the reference is inside a loop (self-reuse, Algorithm 2). So, our implementation analyzes the cache lines accessed in the loop by interleaving references (function *conflictingAccessesInLoop*) in order to see how many of them may be mapped to the set (or sets) used by the self-reuse reference. If the number of possible conflicting cache lines is lower than the LRU minimal cache lifespan [27] (i.e., lower than the number of ways), a hit is guaranteed (FM category for constant addresses, and KM for sequential accesses). Otherwise, this reference is categorized as NC.

If a reference has *group-reuse*, reuse information provides its last dominant reference with group reuse (reference *domref* in Algorithm 1). *domref* dominates *ref* if every path to *ref* goes through *domref*. So, *domref* is the last previous reference that has compulsorily accessed the same data. Hence, a cache hit is guaranteed if the accesses between them do not evict the content to be reused. Similarly as above, our implementation explores the possible paths between each pair of references with group-reuse, retrieving the accessed cache lines that map to the analyzed cache set (function *conflictingAccessesBetween*). If the number of conflicting cache lines is lower than the number of ways, a hit is guaranteed (AH category).

For references with self-reuse within a given loop and group-reuse with respect to another reference dominating this loop (Algorithm 1, line 17), if they present hit on group-reuse (first iteration), this added hit is taken into account in the self-reuse classification performed above. That is, FM are translated to AH, NC to FH, and the k misses in KM are decremented in 1 miss. For simplicity, Algorithm 1 does not show some details, for instance those regarding references in unfeasible paths.

Both Algorithm 1 and 2 use *conflictingAccessesBetween()* and *conflictingAccessesInLoop()* functions to explore the CFG, respectively. Such functions recursively explore the targeted paths in the CFG, either between two references, or between the same reference in different iterations of a loop. Although this targeted brute force exploration is not particularly efficient, notice that both functions work with a perfectly delimited subset of the CFG, and compilers place reusing accesses as close as possible. Also, if the number of conflicting cache lines gets higher than the number of ways (*mls()* function), our implementation terminates the analysis (not show in algorithms 1 and 2). Based on our experiments, performance seems acceptable, as we show in Section 7.5.

3.4.3 ACDC

Classification for the ACDC (Algorithm 4) is much easier, since the ACDC prevents undesired evictions and its behavior depends on the preconfigured data replacement permissions. For references with *group-reuse*, hits are guaranteed (AH category) if any dominant reference with group reuse has been granted data replacement permission. For references *without exploitable group-reuse*, hits are guaranteed if the reference is inside a loop (self-reuse) and has been granted data replacement permission (FM category for constant addresses, and KM for sequential accesses). Otherwise, the reference is classified as NC.

3.4.4 Unlimited size data cache

For comparison purposes, we also analyze an unlimited size data cache. Since no data overlapping can be assumed in the worst case, we resort to the reuse information. That is, accesses without self/-group reuse are assumed to access non-cached memory lines. This would be equivalent to the previous LRU analysis assuming that all reused data hit, or to

Algorithm 1 Algorithm to classify data memory references in a system with a LRU cache.

```
1: for all $ref$ do    # $\forall$ data references in the program
2:   if $ref$ has group reuse then    # group reuse
3:     $domref \leftarrow lastDominantWithGroupReuse(ref)$
4:     if $ref$ is not enclosed in any loop then
5:       if $conflictingAccessesBetween(domref, ref) < mls(LRU, Ways)$ then
6:         $category[ref] \leftarrow$ AH    # single hit
7:       else
8:         $category[ref] \leftarrow$ NC    # single unclassified
9:       end if
10:    else    # $ref$ inside loop
11:      if $loop(ref) = loop(domref)$ then    # both in the same loop
12:        if $conflictingAccessesBetween(domref, ref) < mls(LRU, Ways)$ then
13:          $category[ref] \leftarrow$ AH
14:        else
15:          $category[ref] \leftarrow$ NC
16:        end if
17:      else # reference ($domref$) dominating a loop and reused ($ref$) within it
18:        $category[ref] \leftarrow classifySelfReuse(ref)$
19:        if $category[ref] =$ KM then
20:          $maxMisses[ref] \leftarrow calcLoopMaxMisses(ref)$
21:        end if
22:        if $conflictingAccessesBetween(domref, ref) < mls(LRU, Ways)$ then
23:          if $category[ref] =$ FM then
24:            $category[ref] \leftarrow$ AH
25:          else if $category[ref] =$ KM then
26:            $maxMisses[ref] \leftarrow maxMisses[ref] - 1$
27:            if $maxMisses[ref] = 0$ then
28:              $category[ref] \leftarrow$ AH
29:            end if
30:          else if $category[ref] =$ NC then
31:            $category[ref] \leftarrow$ FH
32:          end if
33:        end if
34:      end if
35:    end if
36:  else    # no group reuse
37:    if $ref$ is inside loop then    # self reuse
38:      $category[ref] \leftarrow classifySelfReuse(ref)$
39:      if $category[ref] =$ KM then
40:        $maxMisses[ref] \leftarrow calcLoopMaxMisses(ref)$
41:      end if
42:    else    # no reuse
43:      $category[ref] \leftarrow$ NC
44:    end if
45:  end if
46: end for
```

Function *conflictingAccessesBetween*(*ref1*, *ref2*) returns the maximum number of conflicting accesses between two references with group reuse with a dominance relation. Function *mls*(*policy*, *ways*) returns the minimal life-span metric in the cache [27].

Algorithm 2 *classifySelfReuse*(*ref*): Fuction to classify a data memory reference inside a loop considering only its self reuse in a system with a LRU cache.

```
1: if $conflictingAccessesInLoop(ref) < mls(LRU, ways)$ then    # not evicted between accesses of $ref$
2:   if $ref$ is constant then    # self-temporal only
3:     return FM
4:   else if $ref$ is array then    # self-spatial reuse
5:     return KM
6:   else if $ref$ is nonlinear then    # reuse not guaranteed
7:     return NC
8:   end if
9: else    # accessed line may be evicted
10:  return NC
11: end if
```

Function *conflictingAccessesInLoop*(*ref*) returns the maximum number of conflicting accesses in the loop enclosing *ref* between two accesses of *ref* in different iterations.

Algorithm 3 *calcLoopMaxMisses*(*ref*): Function to get the potential data misses for *ref* [29].

```
1: if $e_n \in Ker(H_s)$ then # self-spat. (may have self-temp.)
2:   if $e_n \in Ker(H)$ then    # self-temporal
3:     return 1    # access to the same address always
4:   else    # self-spatial reuse
5:     if $lineSize \leq |h_n|$ then    # cache line size $\leq$ stride
6:       return $loopIterations$    # always may miss
7:     else    # exploitable self-spatial reuse
8:       if $h_n > 0$ then    # forward traversal
9:         $firstLineElems \leftarrow lineSize - \left\lfloor \frac{c \mod lineSize}{h_n} \right\rfloor$
10:      else    # backward traversal
11:        $firstLineElems \leftarrow \left\lfloor \frac{c \mod lineSize}{-h_n} \right\rfloor$
12:      end if
13:      return $1 + \left\lceil \frac{loopIterations - firstLineElems}{lineSize/|h_n|} \right\rceil$
14:    end if
15:  end if
16: else    # ref without reuse: may always miss
17:   return $loopIterations$
18: end if
```

Function $Ker(H)$ performs the kernel operation on matrix H, i.e., obtains the set of vectors that are mapped to the null vector by H. Matrix H is the linear transformation of *ref*, h_n is its stride in the enclosing loop, and c is its base address (see Section 3.4.1). $\vec{e}_i$ is a vector with all elements equal to 0 except the one in position i, matrix H_S is H with all elements of its last row replaced by 0, and n is the number of columns of H, i.e. the depth of the nested loops in *ref* [29,35]. Constant *lineSize* represents the size of the cache line, and *loopIterations* is the number of iterations in loop n.

Algorithm 4 Algorithm to classify data memory references in a system with an ACDC cache.

```
1: for all ref do            # ∀ data references in the program
2:   if ref has group reuse and any dominant reference has DRP then   # exploitable group reuse
3:     category[ref] ← AH
4:   else if ref is inside loop and ref has DRP then   # exploitable self reuse
5:     if ref is constant then   # self-temporal reuse only
6:       category[ref] ← FM
7:     else if ref is array then   # self-spatial reuse
8:       category[ref] ← KM
9:       maxMisses[ref] ← calcLoopMaxMisses(ref)
10:    else if ref is nonlinear then   # reuse not guaranteed
11:      category[ref] ← NC
12:    end if
13:  else   # no reuse
14:    category[ref] ← NC
15:  end if
16: end for
```

the previous ACDC analysis assuming that all references have data cache replacement permission. In any case, there would be no replacements in an unlimited size data cache, so writebacks are not considered.

3.4.5 Example of classification

Let us illustrate how the code in Figure 4 would behave depending on the selected data cache and analysis method. For simplicity, let us consider potential misses as misses.

Table 2 shows the category, number of misses, and number of writebacks, for five different systems, namely a system without data cache, two systems with a conventional LRU data cache with at least two ways assuming two different analysis for data references (persistency/address-only and reuse/pattern), a system with an ACDC with at least three entries, and a system with a data cache of unlimited size.

Without cache (column "No cache"), all accesses go to main memory, so there is no need of categorization. Also it presents no writebacks. For an address-only LRU analysis (Figure 5a), in column "LRU-addr", all accessed addresses are unknown, so they are classified as NC. In addition, the data modified by the store instruction must be written to memory on every access.

If we consider a pattern-based LRU analysis (Figure 5b), in column "LRU-pattern", categories are much more accurate than those in the address-only LRU analysis. The *ld B* is still categorized as NC due to the interleaving accesses. On the other hand, the reuse information allows an AH categorization of *st A*, since the number of possible conflicting data lines between this reference and *ld A* (just one, brought by *ld C*) is lower than the number of cache ways (assuming a cache with 2 ways at least). Furthermore, both *ld A* and *ld C* are classified as KM, with $k = n/L$ in its enclosing loop (j) as detailed above. Since loop j is inside two nested loops, both iterating n times, the maximum number of misses for both KM references is bounded to $n^2 \cdot k = n^3/L$. Moreover, the store to A sets the dirty flag for the data brought by the load of A, so it triggers as many writebacks as misses by *ld A*. We associate writebacks with the reference that brings the data from memory, but they could be associated with the reference that sets the dirty flag, or the reference that evicts the cache line.

Let us now consider an ACDC with at least 3 entries for data replacement permissions (column "ACDC"). We assume that all references but the store are granted DRP, that is, they have an exclusively assigned data cache line to replace, and no other instruction can evict it. The store to A reuses the data cached by *ld A*, so it always hits. For *ld A* and *ld C*, ACDC performs as the LRU-pattern case, that is, KM categories with no more than n^3/L misses. For the *ld B* reference, since it has an associated cache line, interleaving accesses cannot evict it, so it is also classified as KM with $k = n/L$. Since its enclosing loop is reached n times in loop i, *ld B* is bounded to n^2/L misses.

Finally, let us discuss our estimation for an unlimited size data cache (column "Unlimited size"). Reused content (*st A*) always hits and, given the unlimited size, there would be no replacements/writebacks. Also, all sequential accesses miss as many times as cache lines the data structures occupy, that is, KM with $k = n/L$ per matrix row traversal. As above, our analysis multiplies those misses by the number of times they occur (n for *ld B*, and n^2 for *ld A* and *ld C*). However, notice that this may involve overestimations, as can be seen for *ld A* and *ld C*. Since matrices A and C occupy n^2/L cache lines, ideally there cannot be more than n^2/L misses, but our analysis estimates n^3/L misses. To try to avoid such overestimation, an address space analysis of data structures plus an out-of-bounds analysis on references would be required. However, notice that such overestimation for the unlimited size cache does not affect its value as a baseline, since it is still a lower bound for our analyzed caches. As stated above, such lower bound corresponds to an LRU-pattern that always takes profit of the existing reuse, or an ACDC

Table 2: Estimated number of misses and writebacks in the worst case for different data caches for the matrix multiplication code in Figure 4, assuming aligned non-overlapping matrices. The considered LRU data cache has 2 ways at least, and the ACDC may hold 3 data replacement permissions at least.

	No cache	LRU-addr			LRU-pattern			ACDC				Unlimited size		
Ref.	Access	Cat	Miss	WB	Cat	Miss	WB	DRP	Cat	Miss	WB	Cat	Miss	WB
ld B	n^2	NC	n^2	0	NC	n^2	0	Yes	KM	n^2/L	0	KM	n^2/L	0
ld A	n^3	NC	n^3	0	KM	n^3/L	n^3/L	Yes	KM	n^3/L	n^3/L	KM	n^3/L	0
ld C	n^3	NC	n^3	0	KM	n^3/L	0	Yes	KM	n^3/L	0	KM	n^3/L	0
st A	n^3	NC	n^3	n^3	AH	0	0	No	AH	0	0	AH	0	0
Total	$3n^3 + n^2$		$4n^3 + n^2$			$\frac{3}{L}n^3 + n^2$				$\frac{3}{L}n^3 + \frac{1}{L}n^2$			$\frac{2}{L}n^3 + \frac{1}{L}n^2$	

with unlimited DRPs, in both cases without writebacks.

In this example, the pattern-based LRU analysis gives the highest estimate of the number of memory accesses in the worst case, even worse than a system without cache. ACDC performs less memory accesses than LRU-pattern in the worst case, but note that it has a limited amount of DRPs to grant. If such number is not enough, references that would benefit from DRP will be categorized as NC. These details are considered in the IPET model.

4 IPET integration

The last part of the WCET analysis commonly involves generating an integer linear programming (ILP) model to obtain the WCET bound. The Implicit Path Enumeration Technique (IPET) defines a flow-based ILP model of the control-flow graph (CFG) by means of a variable x_i for each basic block i in the CFG, and variables d_e for the CFG edges e between basic blocks, both representing the number of times that they are traversed [23]. Two virtual basic blocks *start* and *end* are also linked to the CFG, with their corresponding x variables set to 1. The execution time is defined as:

$$ET = \sum_i c_i \cdot x_i \tag{1}$$

where c_i is the constant cost of traversing basic block i a single time. Then, the WCET bound is obtained by maximizing eq. 1. In order to consider a LRU data cache, the data memory latency cost is removed from c_i, and other variables and constraints are added [23]. In our case, the data cache is not modeled as defined by previous studies, but based on our previous categories, as we detail below. Table 3 describes the original IPET variables and constants, and also the new ones used in our approach.

Table 3: Variables and constants of our IPET proposal.

Variable	Description
d_e	Times that edge e is traversed (original IPET)
dac	Cumulative data access cost in the program
dh_{ref}	Times that data reference *ref* hits
dm_{ref}	Times that data reference *ref* misses
wb_{ref}	Times that data cached by reference *ref* are written back
x_i	Times that basic block i is traversed (original IPET)
Constant	**Description**
c_i	Cost of traversing basic block i (original IPET), without including data memory access costs
k	Constant associated with a given *ref* classified as KM
hc	Data hit cost
mc	Data miss cost
wbc	Writeback cost

In order to integrate our proposal, we represent the total data access costs in the program as a new variable dac, to be added to eq. 1:

$$ET = dac + \sum_i c_i \cdot x_i$$

This variable is the sum of the number of possible occurrences (data hits dh, data misses dm, and writebacks wb) times their constant cost (hit cost hc, miss cost *mc*, and writeback cost *wbc*) for each reference *ref* in the program:

$$dac = \sum_{ref} hc \cdot dh_{ref} + mc \cdot dm_{ref} + wbc \cdot wb_{ref}$$

Then, such occurrences (dh_{ref}, dm_{ref}, and wb_{ref}) are constrained using the original IPET variables x and d as described below. For a clearer notation, let us define the following functions: $BB(ref)$ returns the basic block i where reference *ref* is located, $loop(ref)$ returns the loop l enclosing reference *ref*, and $EE(l)$ returns the set of entry edges of loop l, that is, the set of edges reaching loop l that are not back-edges (given two basic block nodes a, b from a control-flow graph, a back-edge is an edge $a \to b$ whose head b dominates its tail a [2], i.e., all edges that enter the loop header b from the loop body are back-edges). In general, the sum of data hits and misses of a given reference equals the number of times its corresponding basic block is traversed:

$$dh_{ref} + dm_{ref} = x_{BB(ref)} \qquad (2)$$

However, some instruction sets (e.g. ARM) provide predicated load/store instructions. That is, load/-store operations together with a condition so that, when the instruction is executed, the memory access is performed only if the condition is true. In such case eq. 2 must be set as an $\leq$ inequality.

For each reference, we bound its number of misses depending on its category, as follows.

$$0 \leq dm_{ref} \leq \begin{cases} 0 & \text{if } ref \text{ is AH,} \\ \sum_{d \in EE(loop(ref))} d & \text{if } ref \text{ is FM,} \\ \sum_{d \in EE(loop(ref))} k \cdot d & \text{if } ref \text{ is KM,} \\ x_{BB(ref)} - 1 & \text{if } ref \text{ is FH,} \\ x_{BB(ref)} & \text{if } ref \text{ is NC.} \end{cases}$$

As it can be seen, such constraints are a straightforward translation of the categories in Table 1. For AH, no misses are possible. For NC, all accesses ($x_{BB(ref)}$) may miss. For FH, all but one access ($x_{BB(ref)} - 1$) may miss. Notice that in FH cases with $x_{BB(ref)} = 0$ the model is infeasible. If this happens, FH cases can be safely modeled as NC. For FM and KM, the actual misses must be equal or lower than the maximum possible number of misses (1 for FM and k for KM) in the corresponding enclosing loop (reached $\sum d$ times, $\forall\ d \in EE(loop(ref))$). The accesses to scalar variables with group reuse in the ACDC are simpler than our generic FM category, since they can present a single miss at most. So, its precision in the IPET representation can be improved by $0 \leq dm_{ref} \leq 1$, which also simplifies the analysis.

In a similar way, the number of writebacks can be bounded. As outlined in Table 2, the number of writebacks, if any, is equal to the number of misses in the updated data structure. So, we associate the writebacks to the reference that brings the content that will be eventually evicted. The number of writebacks depends on the type of cache. For LRU, writebacks will eventually occur for store misses, and also for load misses that are modified before being evicted. This second situation occurs for loads with both a dominance relation and a group reuse relation to stores, without misses to the reused data between them. In any case, a single writeback per data line occurs, even if the same line is rewritten multiple times.

$$wb_{ref} = \begin{cases} dm_{ref} & \text{if } ref \text{ is store NC/FH/KM/FM,} \\ dm_{ref} & \text{if } ref \text{ is load NC/KM/FM followed (dominating group reuse) by zero or more loads AH, followed by a store AH,} \\ 1 + dm_{ref} & \text{if } ref \text{ is load FH followed (dominating group reuse) by zero or more loads AH, followed by a store AH,} \\ 0 & \text{otherwise.} \end{cases}$$

For the ACDC, only the instructions with data replacement permission may replace content in cache, so only them can cause writebacks. Notice that the FH category is not possible for the ACDC.

$$wb_{ref} = \begin{cases} dm_{ref} & \text{if } ref \text{ is store NC/KM/FM with DRP,} \\ dm_{ref} & \text{if } ref \text{ is load NC/KM/FM with DRP followed (dominating group reuse) by zero or more loads AH, followed by a store AH,} \\ 0 & \text{otherwise.} \end{cases}$$

5 Safety of our approach

Our approach is essentially a series of transformations, from the data reuse facts of the references in the CFG, to an ILP model to be solved. Safety of the data reuse facts is guaranteed by its own analysis based on Abstract Interpretation [11, 28]. Our proposed algorithms classify these reuse facts into categories (Section 3.4). These algorithms are based on the algebraic foundations of the well known reuse theory [35], briefly introduced in Section 3.4.1. For each situation, these algorithms always assume the most loose category (that with less guaranteed information). Such behavior guarantees their safety.

IPET is the most widely used method to model the CFG for a static WCET analysis. It declares the relation between basic blocks, and lets the solver to maximize the objective function of the model, i.e., to obtain the WCET. Our proposal to integrate our categories in the IPET model also follows this approach: For each reference, we declare its possible outcomes (hits plus misses, and writebacks), bound them to the number of times that each reference is executed, and bound the misses and writebacks according to its corresponding category. With such relations and bounds, the solver explores the integer variables (number of hits, misses, etc.) and maximizes the WCET function.

As it can be seen, all parts of our proposal follow the standard safety guidelines in the field. Additionally, below we evaluate two baselines (always hit, and unlimited size data cache) to further validate our results.

6 ACDC configuration

The ACDC is a configurable component [29]. As such, it requires an adequate configuration, dependent on the task to be run. This configuration consists of the set of program counters (PCs) of load-/store instructions with granted data cache replacement permission (DRP). As detailed in Section 3.4, classification of data references into categories is based on DRPs. However, selecting a set of PCs that effectively minimizes the WCET bound when granted DRPs is hard. An existing proposal generates the optimal ACDC configuration for single-path tasks [29]. Since our tested benchmarks are not single-path, we propose a feasible heuristic method to generate the ACDC configuration. Essentially, we perform an always miss WCET analysis without writebacks (equivalent to having an ACDC with no DRPs) and, for each instruction (PC) suitable to be granted DRP, we estimate the benefit b_{PC} that the DRP would provide. Instructions suitable to be granted DRP are those which do not reuse data from other loads/stores, and their data are reused by itself or others. As shown in the following equation, to estimate the benefit of granting DRP to one of these candidate instructions, we must add a cost (positive number of cycles) with a benefit estimate (negative numbers). The cost corresponds to writing once (*preload*) the PC of the selected instruction in the AC part. The benefits can be estimated by adding the savings with respect to the always-miss model due to: a) the access to the data of the selected instruction itself ($access_{ref}$), b) the access of other load/store instructions that reuse the same data ($reuse_{ref}$), and c) the writebacks ($writeback_{ref}$).

$$b_{PC} = preload + \sum_{ref\ in\ PC} access_{ref} + reuse_{ref} + writeback_{ref}$$

The benefit in the access cost of the reference ($access_{ref}$) can be calculated depending on the access type. A reference outside loops is forcefully scalar, and if it is being considered for DRP means that it does not reuse the data of other loads/stores. In such case, it will miss, as already considered for the always-miss model, so the benefit is 0. For references inside loops, the benefit in the access cost of the load/store can be calculated depending on whether it is a scalar reference (self-temporal reuse) or an array reference (self-spatial reuse). In both cases we calculate always a hit cost instead of the original miss cost for all accesses (dm_{ref}) of this instruction, and then revert to miss cost the potential misses, that is, one miss for scalars and, for arrays, k misses each time its enclosing loop is reached. Other cases are not suitable to be granted DRP. Summarizing:

$$access_{ref} = \begin{cases} 0 \quad \text{if } ref \text{ is (scalar) outside loop,} \\ (hc - mc) \cdot dm_{ref} + (mc - hc) \\ \quad \text{if } ref \text{ is scalar inside loop,} \\ (hc - mc) \cdot dm_{ref} + (mc - hc) \cdot \sum_{d \in EE(loop(ref))} k \cdot d \\ \quad \text{if } ref \text{ is array inside loop.} \end{cases}$$

For the references r reusing the data brought by ref (group reuse relation and dominance relation), the benefit implies considering cache hit costs instead of miss costs for all their accesses (dm_r).

$$reuse_{ref} = (hc - mc) \cdot \sum_{r\ reusing\ ref} dm_r$$

The costs of writebacks, not present in always-miss, can be calculated as above, depending on whether the working data set is modified or not.

$$writeback_{ref} = \begin{cases} wbc \cdot \sum_{d \in EE(loop(ref))} k \cdot d & \text{if } ref \text{ has group reuse stores,} \\ 0 & \text{otherwise.} \end{cases}$$

Previous constraints do not affect the maximization objective for the always-miss system, but just calculate a b_{PC} value for each load/store instruction.

Once the model is solved, we select as many candidates as entries in the target ACDC, ordered by their calculated b_{PC} (the lower the better, considering negative values only). It is important to notice that such a selection is not necessarily optimal, since DRPs may affect the worst path, and their effect may depend on the other DRPs. Nevertheless, such estimations provide good results with a simple analysis.

7 Evaluation

In this section we describe the considered target hardware architecture and the benchmarks, and also discuss our experiments and results.

7.1 Target hardware

The target instruction set architecture considered in our experiments is ARMv7 with instructions of 4 bytes. We assume a memory architecture consisting of separated L1 instruction and data caches, both below RAM modules as main memory. At present, a common general purpose L1 cache configuration might be 8-way set-associative, with 64 sets, lines of 64 bytes, and PLRU replacement [1]. In this study we assume LRU instruction and data caches with the same configuration, keeping the number of sets to 64 and varying the number of ways between 4 and 32 (each cache stores between 16 and 128 KiB of instructions/data). This includes configurations with less ways than current general purpose processors, which may be currently dominant in the embedded domain, and also configurations with more ways, to provide an insight of future trends. Notice that PLRU is not adequate for real-time systems, so any PLRU cache would imply larger WCETs than LRU. As an alternative to the data cache, we also test the ACDC cache [29]. As described in Section 2, it works by associating specific data cache lines to preconfigured load/store instructions. The selection of such associations is detailed in Section 6. So, these instructions perform replacements in a controlled way (no other instruction can evict the content in the associated cache line), and other instructions are forced to bypass the cache in case of miss. We assume the same number of entries for the AC (storing PCs with data cache replacement permission) and DC (holding the data), varying from 4 to 32, as the number of ways in the LRU cache. This means 4 to 32 instructions with data replacement permission to their associated data cache line. So, the ACDC may store between 256 and 2048 bytes of data, that is, 64 times less data than its LRU counterpart, since it has no sets (see Figure 2).

In order to focus on data, we model an instruction cache with unlimited size. This is done by limiting the number of misses of each instruction memory line to 1, although previous LRU size suffices to completely hold any of the tested benchmarks. We assume a memory latency of 13 cycles both for instructions and data, which is a realistic value for main memories such as the Automotive DRAM MT46V16M16 [24] clocked at 100 MHz, and has been used in previous studies [32].

We assume a typical 5-stage pipeline (fetch FE, decode DE, execute EX, memory MEM, writeback WB) performing ideally (ideal branch prediction and 1 cycle/stage), except for memory operations (instruction fetch and data memory transfers). FE stage takes 1 cycle for an instruction cache hit, and 14 for a miss (look-up plus memory transfer). Regarding data memory accesses, the address computation is performed in the EX stage (1 cycle), and the data cache look-up/hit for loads is performed in the MEM stage (1 cycle). If the target address is not cached, a memory transfer is triggered, forcing the pipeline to halt until the memory transfer is completed. For stores, we assume the same procedure (although located in the WB stage): 1 cycle to reach the data cache, and 13 additional cycles if the line to write to must be brought from memory (fetch on write-miss policy). For stores with write-around (ACDC only), in the WB stage, we also assume 1+13 cycles, even though a cycle could be saved by performing the AC look-up in the MEM stage. So, for an instruction performing a single memory access, given that the pipeline hides the address computation and the data cache look-up, only accesses that require a memory transfer suffer a penalty of 13 cycles (memory latency). If any missing access evicts a dirty cache line, a writeback is triggered, and its corresponding memory latency is also added to the completion time of the instruction. Finally, in case of push/pop instructions with multiple data memory accesses, the corresponding stage is repeated as many times as requested accesses, and the cost of each access is computed independently, i.e., no burst memory transfers are considered. Table 4 summarizes these costs. It is important to notice that, on data misses, a system with data cache performs worse than without it, specially if the missed data replaces a dirty cache line.

Although we assume a simple pipeline, more complex pipelines can be integrated into IPET (e.g., [4]).

In such case, our bounds on the number of misses would be applied to the specific pipeline constraints in the model.

7.2 Benchmarks

Table 5 shows the benchmarks used in our experiments, compiled by gcc 9.2.1, from the TACLeBench suite [12]. Recursion has not been addressed in this work, so recursive benchmarks have been discarded. We use *angr* version 9.0.4663 to extract and process the CFGs [30]. Although our proposal has no restrictions regarding the CFG, it must be taken into account that angr is in active development stage, and it may decode some instructions incorrectly. In the cases that such errors result in invalid CFGs, the corresponding benchmarks have been discarded. Also, benchmarks *deg2rad* and *rijndael_dec* have been discarded because their results are almost identical to those of *rad2deg* and *rijndael_enc*, respectively. Benchmark *cover* has also been discarded due to the fact that it has very few memory references and none of them inside a loop. For each one of the binaries, flow information (flow facts) has been manually set based on the annotations in the source code, carefully studying the effect of compiler optimizations. Nevertheless, existing loop bound analysis methods could be used [7, 22]. Table 5 shows the considered benchmarks, along with an estimation of the number of data memory accesses in the estimated WCET case for each compiler optimization level, discussed below. Most of these benchmarks contain procedures, which can be transformed in different ways, depending on compiler optimizations, such as inlining, cloning, or specialization for constant parameters.

In order to provide some insight into the data complexity, Figure 6 shows the number of static load/store instructions for each benchmark and optimization level. Benchmarks are ordered by the number of such instructions when compiled without optimizations. As it can be seen, benchmarks on the left side contain very few memory instructions, whereas those on the right have up to two orders of magnitude more load/store instructions. In general, binary codes compiled without optimizations have redundant loads/stores, which are removed when optimizing. A few exceptions can be found with optimization level 3, which tries to unroll loops with few iterations. If so, loads/stores inside these loops are replicated, as can be seen for instance in *complex_updates* and *cjpeg_transupp*. All figures in this section follow this ordering for the benchmarks.

7.3 Data cache hit ratio

In this section we present the hit ratio for different cache configurations. Data cache hit ratio is the percentage of data hits out of the total data accesses. So, it is the most direct measure of the effectiveness of the cache.

Since Figure 6 shows benchmarks with very few memory instructions, it is important to verify that the number of performed memory accesses is reasonably high. This is a complex problem on its own, since the number of performed memory accesses depends on the taken path, and the path associated to the WCET bound may depend on factors such as the tested data cache. In order to provide a path-independent context, we define the number of data memory accesses in the estimated WCET case (NMAWC) as:

$$NMAWC = \frac{WCET_{NC} - WCET_{AH}}{MemoryLatency - HitCost}$$

where $WCET_{NC}$ and $WCET_{AH}$ refer to the WCET bounds of systems with no data cache and always hit on data, respectively, and *MemoryLatency* and *HitCost* are the costs of accessing data for the previous systems. That is, $MemoryLatency = mc - hc$, and $HitCost = hc$. Specific data caches would present a number of memory accesses between these two opposite baselines, so the NMAWC provides an insight of such value. Also, when changing parameters, there is very little variation in the path associated to the estimated WCET, since usually the task to perform does not change. The obtained NMAWC values can be seen in Table 5, as an absolute value for O0 and as a percentage with respect to O0 for optimized binaries. It can be seen that optimizations reduce very much the number of data memory accesses. In a few cases, the O3 versions of the binaries increase the number of estimated accesses, which is especially noticeable in *g723_enc*, *ludcmp*, and *rijndael_enc*. The reason is the aggressive code transformations carried out under the O3 flag, such as vectorization, which usually requires loop cloning to deal with remainders of the iteration space. For all the cloned loops, we conservatively keep the bounds specified by TACLeBench. Hence, if they contain data references, the estimated accesses in the worst case may increase.

In systems with one level of cache memory, the data cache hit ratio is calculated by counting as hits all accesses (ld/st) that do not require communication with the off-chip memory. Note, however, that writeback caches, while preventing some memory transfers, can

Table 4: Timing (cycles) considered in data cache operations for the corresponding data access pipeline stage, assuming 1 cycle for cache look-up and 13 cycles of memory latency.

Operation (cache type)	Cost of stage
Cache hit (LRU/ACDC)	1
Load/store (no cache)	13
Cache miss with replacement (LRU/ACDC)	1+13
Cache miss without replacement (ACDC)	1+13
Cache write-around store (ACDC)	1+13
Cache miss with repl. and writeback (LRU/ACDC)	1+13+13
Multiple access (push/pop instructions)	Sum of each access

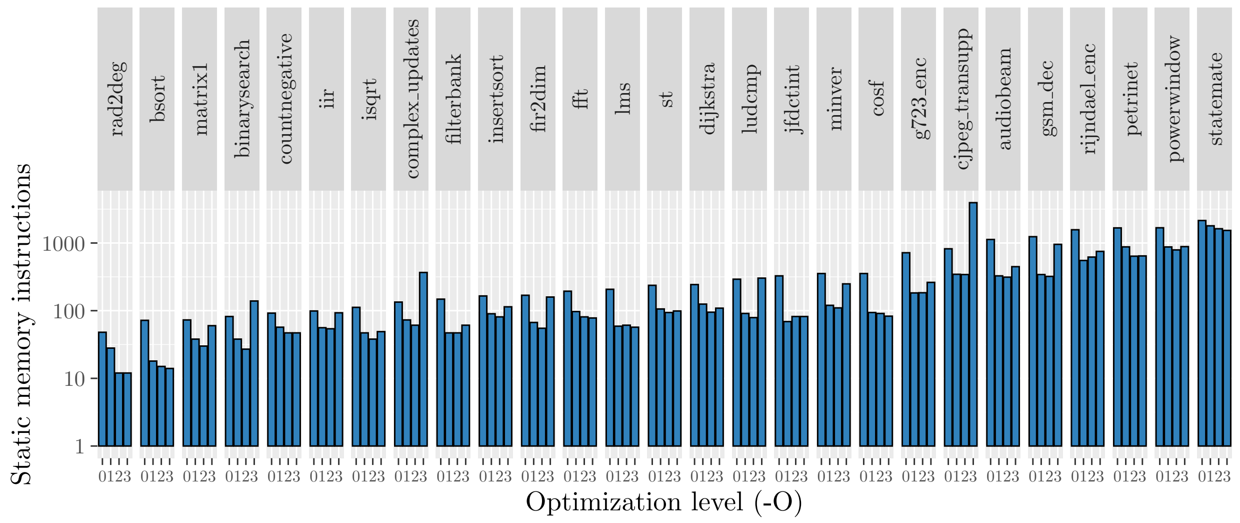


Figure 6: Number of static load/store instructions.

Table 5: Benchmarks, and estimation of their number of data memory accesses in the estimated WCET case.

Name	O0	O1 (%)	O2 (%)	O3 (%)
audiobeam	225681	31.55	29.38	29.49
binarysearch	568	49.35	24.19	24.51
bsort	4039	16.43	13.96	16.13
cjpeg_transupp	32938	23.63	17.34	51.74
complex_updates	1356	35.53	34.99	25.66
cosf	18576	20.50	19.00	13.54
countnegative	503	44.40	28.26	28.26
dijkstra	15482	101.32	42.79	26.33
fft	81591	51.19	32.70	34.94
filterbank	8172	53.61	23.82	29.45
fir2dim	3226	33.10	32.76	34.85
g723_enc	26136	43.54	53.11	177.76
gsm_dec	14999	42.39	31.33	89.16
iir	1350	30.76	30.49	30.49
insertsort	620	35.42	34.08	77.38
isqrt	1309	68.08	59.13	23.75
jfdctint	3004	15.58	20.83	20.83
lms	4413	55.22	46.81	19.33
ludcmp	803	31.38	25.29	106.09
matrix1	2754	25.03	22.22	34.82
minver	663	44.78	44.37	66.34
petrinet	1615	99.71	72.74	72.63
powerwindow	1846228	49.47	48.34	47.70
rad2deg	7320	0.38	0.15	0.15
rijndael_enc	511469	29.52	29.71	169.28
st	102432	23.56	12.72	12.70
statemate	101685	66.62	57.53	55.54

also add additional ones. This occurs when a dirty line is evicted from the cache, which requires writing the modified line to memory, usually without stalling the processor, through a copyback buffer operating in the background. However, notice that a write operation from the copyback buffer uses both the data bus and the corresponding memory banks. So, if there are other memory instructions nearby, such operation does stall the pipeline. Also, stalls occur if the copyback buffer is full, which may depend on the previously taken paths. Since representing such details as an ILP model may be unfeasible, a safe approach is to assume a copyback buffer that always stalls the pipeline. Thus, in order to have a metric that also takes into account the copyback overhead, we propose the following. First, we consider the classical data miss ratio, that is, number of misses out of number of explicit accesses (hits plus misses). Then, we add the writeback ratio (number writebacks out of number of hits plus misses) to previous value, in order to get a memory transfer ratio (average number of lines read or written to memory per memory reference). Finally, we use 1 minus the previous memory transfer ratio to obtain an *effective* data hit ratio (EDHR), i.e., the average number of accesses being serviced within cache time per reference, without accessing memory to read or write lines.

$$EDHR = 1 - \frac{misses + writebacks}{hits + misses} = \frac{hits - writebacks}{hits + misses}$$

Notice that EDHR may result in negative values. Positive values indicate that the cache is effectively saving memory accesses. On the other hand, negative values mean that the number of saved accesses does not compensate the extra memory transfers for writebacks. As a baseline, the EDHR of a system without data cache would be 0. We estimate the EDHR of the estimated WCET case from the solved IPET model:

$$EDHR = \frac{\sum_{ref} dh_{ref} - wb_{ref}}{\sum_{ref} dh_{ref} + dm_{ref}} \tag{3}$$

We test a system with the following data caches: LRU data cache based on a persistence analysis (LRU-addr) [32], LRU data cache based on a pattern analysis (LRU-pattern), and ACDC. Remember that the persistence analyses, which may be considered the state-of-the-art competitive baseline, consider just known constant addresses in memory references, whereas the reuse analysis (used in LRU-pattern and ACDC) also provides information regarding access patterns and reuse.

Figure 7 shows the effective data hit ratio in the estimated WCET case (EDHR, eq. 3), in percentage, for ACDC and LRU caches with 4 to 32 ways. That is, the ACDC has a data storage capacity of 256 to 2048 B, and the LRU (64 sets) can store from 16 to 128 KiB. Values of 100 represent the unreachable always-hit case, and values of 0 represent the no-cache case (highlighted with dashed lines). This is shown for each benchmark (top tags) and compiler optimization level (right tags).

Let us first focus on the LRU results. Perhaps the most important detail is that many times the effective data hit ratio for the LRU cache is below 0, that is, the bound on the number of data memory accesses with a conventional cache is worse than without cache. This may be surprising, given that it is well known that caches have a high hit ratio in average, and the data workload of the tested benchmarks

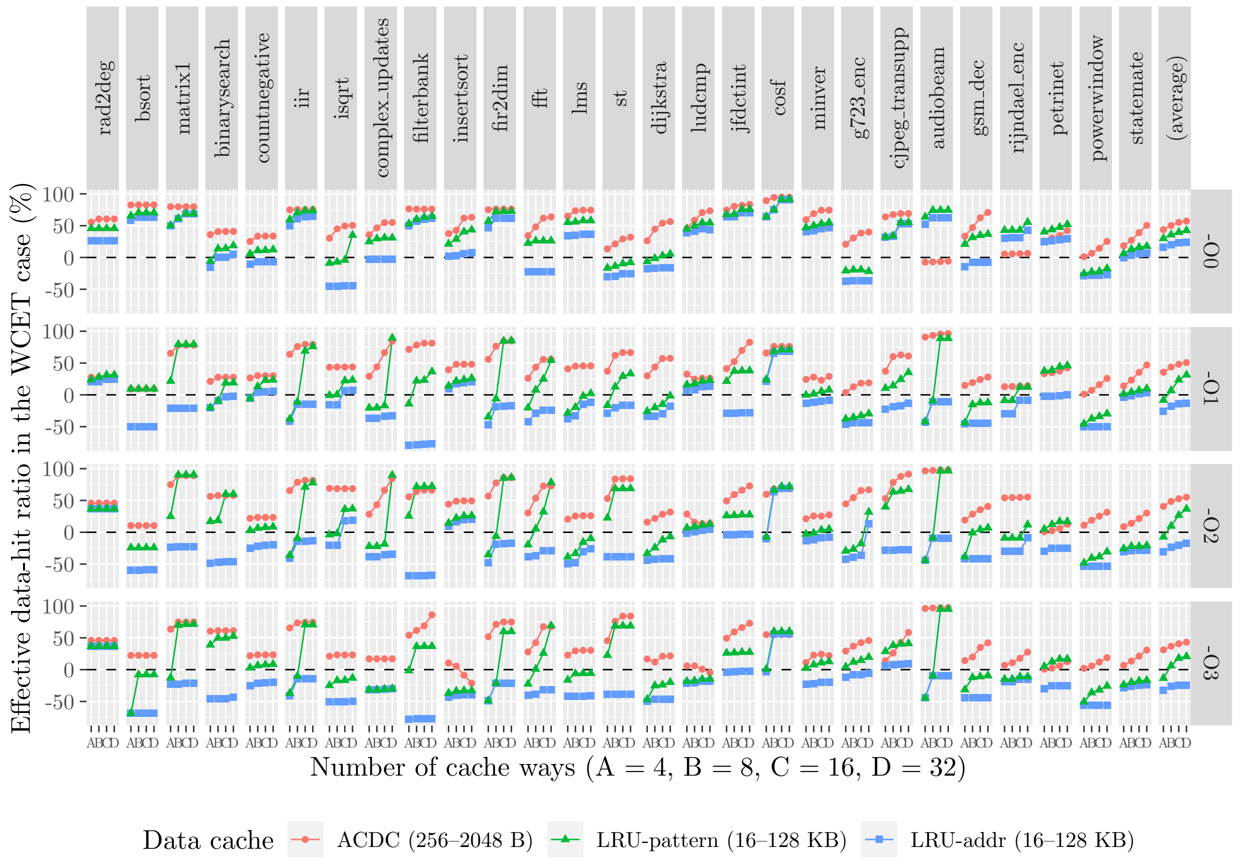


Figure 7: Effective data hit ratio in the estimated WCET case (EDHR, eq. 3) for different data cache ways (4, 8, 16, 32) and optimization levels (-O0, -O1, -O2, -O3).

is not excessively large. Such negative values are due to writebacks, which generate memory transfers that do not occur without data cache (see eq. 3).

Focusing on LRU-addr, even though persistence analysis guarantees an exact result, it is not adequate for unknown or variable memory addresses [32]. Thus, in order to support them, the analysis must assume that the cache line used by anyone of these accesses may be in any set, resulting in the eviction, in the worst case, of whole LRU ways (between 1/4 and 1/32 of the cached data in our experiments).

For LRU-pattern, results are always better than LRU-addr, since it takes profit of the reuse information [28]. With such information, much more hits are guaranteed, even if the specific address of accesses is unknown. However, the pollution they introduce may severely harm the hit ratio, if data may be evicted before being reused. In most cases (and in the average case) it can be seen that the hit ratio increases as the number of ways grows. Notice that, the more the ways in the cache, the more pollution it tolerates. That is, more interleaving accesses may occur between a given access that brings data and another reusing them. In order to confirm that the LRU problems come from accesses to unknown addresses/patterns (which pollute all sets), we have performed the same experiments with a fully-associative LRU with 4 to 32 ways (256 to 2048 B). The results (not shown) are almost identical to having 64 sets. That is, given a fully-associative LRU cache, increasing its capacity by adding sets does not provide significant benefits to the hit ratio in the estimated WCET case.

Regarding the ACDC, in most cases it provides better results than LRU. This is done with a size 64 times smaller than the tested LRU, which effectively confirms that the sets in the LRU are practically useless regarding the WCET bound. The ACDC has several key features that explain its good results.

First, it has no pollution, which is the main drawback of LRU. Since only the instructions with replacement permission can evict contents, all evictions are controlled, meaning that there are no unexpected/undesired evictions. Second, when correctly configured, only worthy instructions are granted permission to replace cache contents, so that instructions with little or no reuse in the worst case bypass the cache. This is specially interesting when considering that cached stores require at least two memory accesses, one for bringing the cache line from memory and eventually another to write it back to memory after its update, so not caching them may be better than the blind cache-anything policy of conventional caches. However, the ACDC also has drawbacks. The first one is its limited size. It is important to notice that the ACDC is preloaded similarly to a locked cache, although the ACDC does not preload data but data replacement permissions. So, it can only grant as many replacement permissions as available entries, that is, between 4 and 32 in our experiments. Such permissions are fixed for the whole benchmark execution, so they may suffice for small benchmarks but not so for large ones. When the size of the ACDC is enough to completely accommodate the benchmark, Figure 7 shows flat results. That is, adding more ways/entries does not provide any improvement (e.g., leftmost benchmarks). On the other hand slopes indicate that there is room for further improvements.

The configurable behavior of the ACDC can also be seen as a drawback, since finding a good configuration is not easy. Although our methodology for obtaining such configuration makes the ACDC results better than LRU in general, they are not optimal. This can be clearly seen, for instance, in *insertsort-O3*, where the ACDC hit ratio decreases as the ACDC capacity grows. This cannot occur for optimal configurations, which demonstrates that our heuristic configurations for 8, 16, and 32 ways are not adequate for this binary. Similar situations appear in some configurations for *dijkstra*, *ludcmp*, and *minver*. Also, particularly bad results such as those found for *audiobeam-O0* and *rijndael_enc-O0* seem to suggest that better configurations are possible. Indeed, we have obtained better results by manually setting the data replacement permissions in some benchmarks (not shown).

7.4 WCET

Although previous section analyzes the improvements directly provided by the data cache, a significant part of the WCET is related to the instruction flow. So, it is required to study how previous results actually impact the WCET. In this section we focus on the results for data caches with 8 ways. Currently, such configuration is broadly used by general purpose commercial processors (i.e. Intel and AMD) in their L1 cache memories, so future embedded processors are likely to use similar caches. Results for 4, 16, and 32 ways present similar trends.

Figure 8 shows the computed WCET bound with respect to the no-cache WCET bound compiled without optimizations, for each binary (both with and without optimizations). Results are grouped by benchmark (top tags) and cache type (right tags), namely ACDC (512 B), LRU-pattern (32 KiB), and LRU-addr (32 KiB). For each optimization level, it also shows the no-cache WCET bound (diamond mark), the unreachable always-hit WCET bound (square mark), and the estimation of the WCET bound for an unlimited size data cache ($\times$ mark). This unlimited size baseline is computed as described in Section 3.4, and provides a lower bound that might be reachable (unlike the always-hit bound). Finally, the fill color of bars represents the benefit of the obtained WCET bound, that is, how close it is from the no-cache WCET bound (0% benefit) to the unlimited size baseline (100% benefit), or how much it gets worse (negative benefits, truncated to -100%).

In order to prevent bad results from shrinking the most interesting part of Figure 8, we let these values go beyond the represented area. The no-cache WCET bounds for these cases are 177% for *g723_enc-O3*, and 169% for *rijndael_enc-O3*. Notice that these are the benchmarks that, when optimized with O3, show aggressive code transformations that increase their number of memory accesses (see Table 5).

As it can be seen, the no-cache WCET bound for non-optimized benchmarks is always 100%, since this is the baseline WCET bound. Optimized benchmarks present lower WCET bounds for the no-cache system, except for *dijkstra-O1*, *ludcmp-O3*, *g723_enc-O3*, and *rijndael_enc-O3*. In average, compiling with optimizations reduces the no-cache WCET bound to 41.7% (O1), 33.0% (O2), and 49.1% (O3) with respect to not optimizing. Notice that these reductions are due to both instruction and data optimizations. Previous studies pointed out O3 as the best optimization level for the WCET bound [26]. However, such studies focused on instruction caches and assumed always hit on data. Figure 8 also shows the estimated always-hit WCET baseline. Although such a WCET value is unreachable, in general it is useful to know

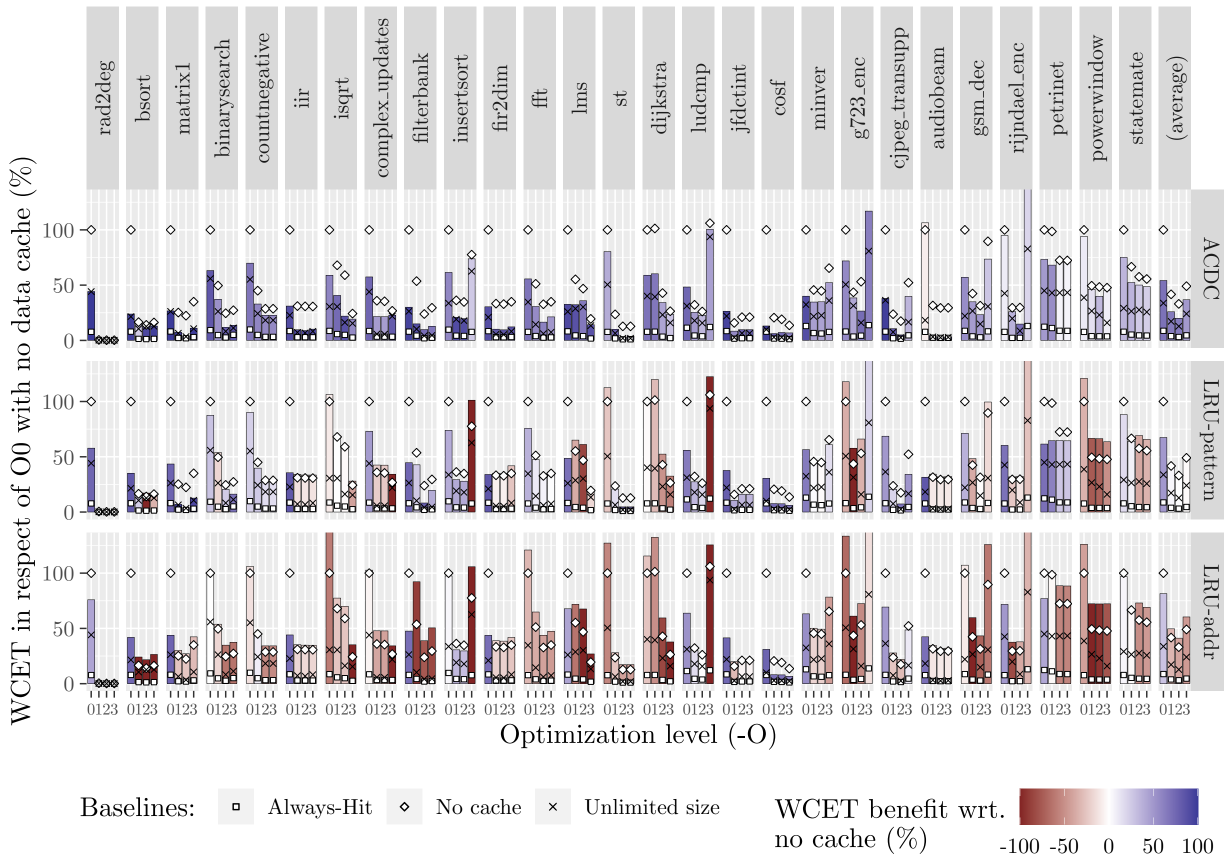


Figure 8: WCET bounds for different optimization levels, and benefit with respect to a system without data cache. ACDC has 8 entries (512 B for data storage) and LRU is an 8-way set-associative cache (64 sets, 32 KiB for data storage).

that no lower WCET values are possible. For instance, we can see that its distance to the no-cache WCET bound is inappreciable in *rad2deg* for optimized codes, due to that they perform just 30 memory accesses for O1, and 12 for O2 and O3. For some other cases (e.g., *cosf*) both ACDC and LRU reach a WCET bound close enough to this unreachable baseline so that it is not worth to look for further improvements.

Regarding the ACDC and LRU results, for non-optimized codes most caches present improvements with respect to no-cache. For optimized binaries, LRU-addr bars are mostly reddish, LRU-pattern have mixed colors (white in average), and ACDC bars are bluish in all cases. This confirms the data cache hit ratio results depicted in Figure 7, where in average the ACDC effective hit ratio is always better than that of LRU, even with just 8 instructions with data replacement permission. Remember that our proposed method for the ACDC configuration is not optimal, so the ACDC might provide results even better than those in Figure 8. Nevertheless, in average the WCET bound obtained with an ACDC is 39.04% shorter than that of an LRU-pattern for the optimization level that results in the shortest WCET bounds (O2). For other optimization levels, the ACDC also provides shorter WCET bounds than LRU-pattern, namely 19.62% (O0), 36.35% (O1), and 25.79% (O3).

7.4.1 General discussion on WCET

Let us discuss our previous baselines and WCET estimations. All of them are generated by state-of-the-art static analysis, so we assume that any introduced overestimation is tight enough for our results to be realistic. Otherwise, differences between them may look closer than they are. In any case, both the baselines and the estimated WCETs are calculated by the same methodology, so any possible overestimation would deviate them in the same way.

Assuming a hard real-time system, our results demonstrate that the ACDC is probably the most adequate data cache. For soft or mixed-criticality real-time systems, if there is a high volume of non-critical tasks and the hardware is dimensioned considering them, LRU (or even PLRU) caches could probably provide better overall results than the ACDC. Nevertheless, considering that the WCET bounds in presence of an LRU cache would be longer, mixed-criticality systems would be forced to overdimension the hardware to accommodate the WCETs of just the most important tasks. Using an ACDC the WCET bounds would be shorter (notice also that the small size of the ACDC allows context switches with a very low penalty), and such a system could be able to schedule all the tasks, not just the most important ones.

In many cases, the cache hierarchy is partitioned in order to isolate tasks and avoid interferences between them [10, 16]. With our proposal, applying set/way partitioning to L1 caches is straightforward, since it would imply to simply assume a cache configuration with less sets/ways. Also, there are studies that extend the may/must analysis to multilevel caches by providing categorizations for each cache level [36]. Our approach could be extended in a similar way. Considering multicore systems with tasks running in parallel, the problem is harder. In such case, cache partitioning would be the only option to avoid interferences between tasks, so it would be required in order to obtain reasonable WCET bounds.

7.5 Analysis time

In this section we discuss the analysis time of our experiments, measured on a 3.36 GHz AMD Ryzen Threadripper 1920X processor.

Table 6 shows the reuse analysis time [28], required to feed our proposal. Since the reuse information it provides is independent of the cache, it must be performed just once for each benchmark.

Figure 9 shows the required time for the WCET analysis for our tested data cache architectures, and also for the system without data cache as a baseline, as boxplots, without including the reuse analysis time (Table 6). Except the no-cache experiments, each bloxplot includes 16 experiments (4 optimization levels times 4 cache configurations). The WCET

Table 6: Data reuse analysis times [28] to generate the data reuse facts required for our proposal, in seconds.

Name	O0	O1	O2	O3
audiobeam	1134	1124	742	40 345
binarysearch	3	2	1	93
bsort	1	1	1	1
cjpeg_transupp	96	66	71	28 298
complex_updates	13	7	7	377
cosf	49	24	29	47
countnegative	8	3	2	2
dijkstra	36	54	162	72
fft	26	62	61	49
filterbank	1	1	2	10
fir2dim	63	19	12	192
g723_enc	3250	1192	4026	3205
gsm_dec	3543	4544	2189	13 757
iir	6	2	3	5
insertsort	39	41	29	29
isqrt	7	5	4	19
jfdctint	149	47	55	56
lms	146	26	45	38
ludcmp	28	49	49	344
matrix1	9	5	3	17
minver	66	200	134	1298
petrinet	138	87	48	50
powerwindow	23 712	4653	17 173	15 727
rad2deg	1	0	0	0
rijndael_enc	14 606	1009	1632	2553
statemate	1157	3751	10 121	5330
st	191	27	15	35

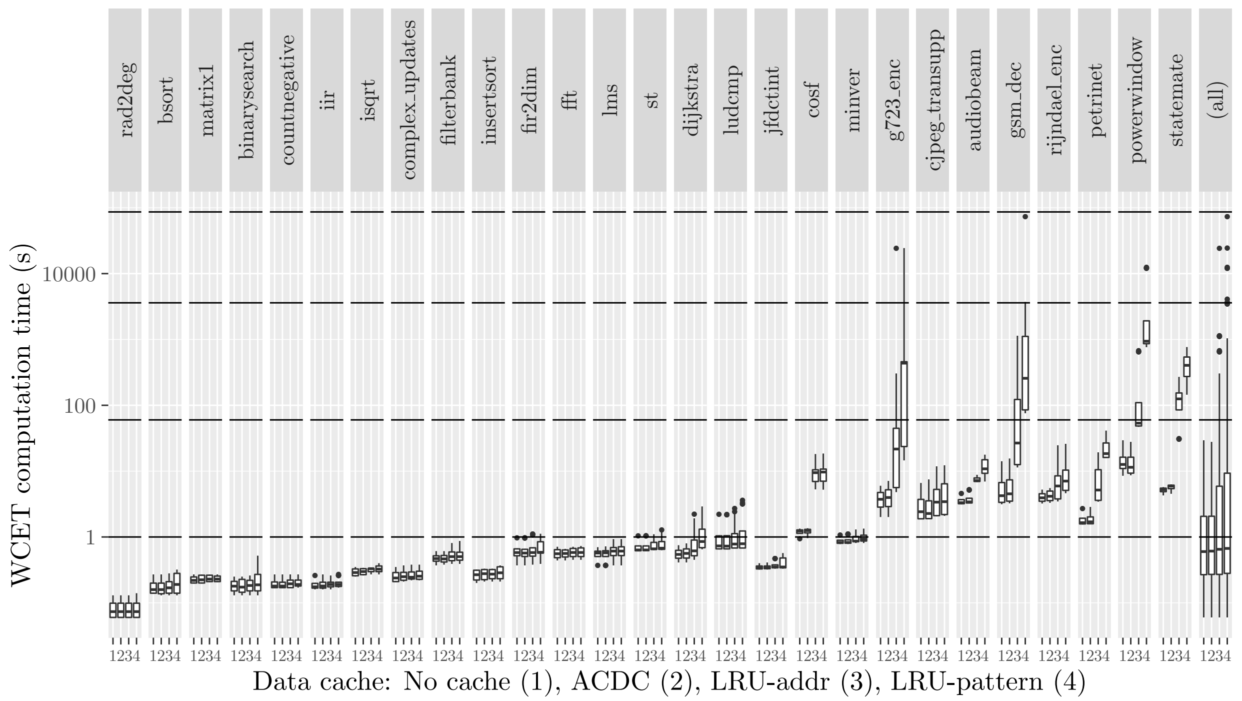


Figure 9: WCET analysis times for the systems without data cache, with ACDC, and with LRU data cache. Horizontal lines mark 1 second, 1 minute, 1 hour, and 1 day.

analysis time includes the processing of the CFG, the generation of the IPET model, and its solving time by lp-solve 5.5.2.5. Both the processing of the CFG and the generation of the IPET model run on python, which may require an execution time around two orders of magnitude longer than an equivalent compiled analyzer. Nevertheless, it can be seen that all times are below one minute, except for the LRU analysis of 4 benchmarks, with only 5 experiments requiring more than one hour. As detailed in Section 3.4, our LRU analysis is implemented as a targeted brute force analysis, i.e., it is not meant to be efficient but to provide accurate results. On the other hand, the analysis time of the system without cache and the system with ACDC is very similar. In both cases, building the IPET model is straightforward, and the whole WCET analysis takes less than 10 seconds except for some experiments for *gsm_dec* and *powerwindow*. Globally, the median of the WCET analyses is under 1 second for all the tested architectures. In most cases, reuse analysis (Table 6, required for our approach) takes more time than our actual WCET analysis.

8 Conclusions

In this paper we propose a generic framework for analyzing the WCET of binary programs in a system with data cache. This framework includes the categories for data references, and how they can be classified depending on the specific cache organization and the reuse information of the task. We apply it to analyze set-associative conventional LRU data caches (writeback with fetch on write-miss and write allocate), an ACDC, an unlimited size data cache, a system without cache, and an ideal always-hit data cache. For the LRU data cache we study both a persistence-based analysis and a reuse-based analysis, and for the ACDC we propose an heuristic method to obtain a good configuration of its data replacement permissions. We also detail how to integrate our data cache categories into an IPET model to obtain the WCET bound.

Our results show that a persistence-based LRU analysis is not adequate for data caches, providing worse hit ratio and WCET bounds than a system without data cache. With a reuse-based analysis, a conventional LRU cache provides a better worst-case performance, but yet similar to a system without cache. In general, the more the ways, the better it performs, since it tolerates more pollution. On the other hand, a high number of sets provides marginal benefits only. Also, writebacks amplify the good/bad results in LRU: writebacks reduce memory accesses when there are few undesired evictions, whereas they increase memory accesses when inconvenient evictions must be assumed. The ACDC provides the best results in general, even with its much smaller size, since its predictable design avoids pollution. In average, the WCET bound obtained with an ACDC is 19.62%, 36.35%, 39.04%, and 25.79% shorter than that of a set-associative conventional LRU data cache for benchmarks compiled with optimization level 0, 1, 2, and 3, respectively. Globally, O2 is the optimization level that results in the shortest WCET bound in average for all our tested cache organizations.

Regarding the required analysis time of our proposal, most of our analyses take less than 10 seconds (below 1 second in median), and only 5 of those implemented by means of targeted brute force take more than one hour. In most cases, reuse analysis (required for our approach) takes more time than our actual WCET analysis.

References

[1] A. Abel and J. Reineke. Reverse engineering of cache replacement policies in intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 141–142. IEEE Computer Society, 2014. doi:10.1109/ISPASS.2014.6844475.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 2007.

[3] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. Combining prefetch with instruction cache locking in multitasking real-time systems. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2010, Macau, SAR, China, 23-25 August 2010*, pages 319–328. IEEE Computer Society, 2010. doi:10.1109/RTCSA.2010.8.

[4] Z. Bai, H. Cassé, M. D. Michiel, T. Carle, and C. Rochange. Improving the performance of WCET analysis in the presence of variable latencies. In J. Xue and C. Jung, editors, *Proceedings of the 21st ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2020, London, UK, June 16, 2020*, pages 119–130. ACM, 2020. doi:10.1145/3372799.3394371.

[5] C. Ballabriga and H. Cassé. Improving the first-miss computation in set-associative instruction caches. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 341–350. IEEE Computer Society, 2008. doi:10.1109/ECRTS.2008.34.

[6] T. Blaß, S. Hahn, and J. Reineke. Write-back caches in WCET analysis. In M. Bertogna, editor, *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, volume 76 of *LIPIcs*, pages 26:1–26:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECRTS.2017.26.

[7] A. Bonenfant, M. de Michiel, and P. Sainrat. oRange: A tool for static loop bound analysis. In *Proceedings of the Workshop on Resource Analysis*, 2008.

[8] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. Exact analysis of the cache behavior of nested loops. In M. Burke and M. L. Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 286–297. ACM, 2001. doi:10.1145/378795.378859.

[9] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In T. P. Baker, editor, *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 47–56. IEEE Computer Society, 2009. doi:10.1109/RTSS.2009.20.

[10] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *2015 IEEE Real-Time Systems Symposium, RTSS*

2015, San Antonio, Texas, USA, December 1-4, 2015, pages 305–316. IEEE Computer Society, 2015. doi:10.1109/RTSS.2015.36.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[12] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In M. Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICS*, pages 2:1–2:10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.WCET.2016.2.

[13] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real Time Syst.*, 17(2-3):131–181, 1999. doi:10.1023/A:1008186323068.

[14] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999. doi:10.1145/325478.325479.

[15] R. Gran, J. Segarra, A. Pedro-Zapater, L. C. Aparicio, V. Viñals, and C. Rodríguez. A predictable hardware to exploit temporal reuse in real-time and embedded systems. *J. Syst. Archit.*, 61(5-6):227–238, 2015. doi:10.1016/j.sysarc.2015.05.001.

[16] Z. Guo, K. Yang, F. Yao, and A. Awad. Intertask cache interference aware partitioned realtime scheduling. In C. Hung, T. Cerný, D. Shin, and A. Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 218–226. ACM, 2020. doi:10.1145/3341105.3374014.

[17] S. Hahn and D. Grund. Relational cache analysis for static timing analysis. In R. Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 102–111. IEEE Computer Society, 2012. doi:10.1109/ECRTS.2012.14.

[18] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 203–212. IEEE Computer Society, 2011. doi:10.1109/RTAS.2011.27.

[19] N. P. Jouppi. Cache write policies and performance. In A. J. Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 191–201. ACM, 1993. doi:10.1145/165123.165154.

[20] C. Lee, K. Lee, J. Hahn, Y. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Trans. Software Eng.*, 27(9):805–826, 2001. doi:10.1109/32.950317.

[21] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 10 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[22] H. Li, I. Puaut, and E. Rohou. Traceability of flow information: Reconciling compiler optimizations and WCET estimation. In M. Jan, B. B. Hedia, J. Goossens, and C. Maiza, editors, *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 97. ACM, 2014. doi:10.1145/2659787.2659805.

[23] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, pages 254–263. IEEE Computer Society, 1996. doi:10.1109/REAL.1996.563722.

[24] I. Micron Technology. Automotive DDR SDRAM MT46V32M8, MT46V16M16. https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf.

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

[26] A. Pedro-Zapater, J. Segarra, R. Gran Tejero, V. Viñals, and C. Rodríguez. Reducing the WCET and analysis time of systems with simple lockable instruction caches. *PLOS ONE*, 15(3):1–21, Mar. 2020. doi:10.1371/journal.pone.0229980.

[27] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real Time Syst.*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

[28] J. Segarra, J. Cortadella, R. G. Tejero, and V. V. Yúfera. Automatic safe data reuse detection for the WCET analysis of systems with data caches. *IEEE Access*, 8:192379–192392, 2020. doi:10.1109/ACCESS.2020.3032145.

[29] J. Segarra, C. Rodríguez, R. Gran, L. C. Aparicio, and V. Viñals. ACDC: small, predictable and high-performance data cache. *ACM Trans. Embed. Comput. Syst.*, 14(2):38:1–38:26, 2015. doi:10.1145/2677093.

[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016. doi:10.1109/SP.2016.17.

[31] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, pages 41–48. IEEE Computer Society, 2005. doi:10.1109/ECRTS.2005.26.

[32] G. Stock, S. Hahn, and J. Reineke. Cache persistence analysis: Finally exact. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 481–494. IEEE, 2019. doi:10.1109/RTSS46320.2019.00049.

[33] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real Time Syst.*, 18(2/3):157–179, 2000. doi:10.1023/A:1008141130870.

[34] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In B. Cheng, S. K. Tripathi, J. Rexford, and W. H. Sanders, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, pages 272–282. ACM, 2003. doi:10.1145/781027.781062.

[35] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In D. S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 30–44. ACM, 1991. doi:10.1145/113445.113449.

[36] Z. Zhang, Z. Guo, and X. D. Koutsoukos. Handling write backs in multi-level cache analysis for WCET estimation. In E. Bini and C. Pagetti, editors, *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017*, pages 208–217. ACM, 2017. doi:10.1145/3139258.3139269.