Automatic Safe Data Reuse Detection for the WCET Analysis of Systems With Data Caches

JUAN SEGARRA [1], **JORDI CORTADELLA** [2], **(Fellow, IEEE), RUBÉN GRAN TEJERO** [1], **AND VÍCTOR VIÑALS-YÚFERA** [1], **(Member, IEEE)**

[1]Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, 50018 Zaragoza, Spain
[2]Computer Science Department, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain

Corresponding author: Juan Segarra (jsegarra@unizar.es)

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, 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 (LRU, locked, ACDC, etc.). In this article, we analyze data reuse (in the worst case) as a property of the program, and thus independent of the data cache. Our analysis method uses Abstract Interpretation on the compiled program to extract, for each static load/store instruction, a linear expression for the address pattern of its data accesses, according to the Loop Nest Data Reuse Theory. Each data access expression is compared to that of prior (dominant) memory instructions to verify whether it presents a guaranteed reuse. Our proposal manages references to scalars, arrays, and non-linear accesses, provides both temporal and spatial reuse information, and does not require the exploration of explicit data access sequences. As a proof of concept we analyze the TACLeBench benchmark suite, showing that most loads/stores present data reuse, and how compiler optimizations affect it. Using a simple hit/miss estimation on our reuse results, the time devoted to data accesses in the worst case is reduced to 27% compared to an always-miss system, equivalent to a data hit ratio of 81%. With compiler optimization, such time is reduced to 6.5%.

INDEX TERMS Real-time, WCET, data-cache, data-reuse.

I. 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. Most WCET analysis methods study the execution flow of the program and its interaction with the hardware, and then build an Integer Linear Programming (ILP) model to solve the problem, either as a flow-based problem [19] or a structure-based problem [3].

The associate editor coordinating the review of this manuscript and approving it for publication was Kaitai Liang [ID].

Analyzing the interactions between the program and the hardware is perhaps the most complex part, since current processors perform many operations with a variable duration in order to improve performance. To mitigate these time-predictability drawbacks of hardware, recent studies propose software-defined architectures [21]. Nevertheless, the memory hierarchy seems an unavoidable problem. A memory hierarchy made up of one or more cache levels takes advantage of 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. Most WCET studies assume just one cache level, although some of them consider a multi-level memory architecture [27].

There are many studies on the worst-case analysis of instruction caches, but data cache analysis is much more complex [20], [23]. This complexity can be seen in common

scenarios such as loops, function calls, and execution-time addressing. In loops, a memory instruction may access different data memory addresses depending on the loop iteration. In functions, memory instructions accessing the local variables of a subroutine 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. With such added complexity, calculating data hits and misses in the worst case analysis is much harder than calculating instruction hits and misses. Furthermore, previous studies show that around half of the WCET comes from data accesses [24].

To the best of our knowledge, all WCET analyses of systems with data caches have focused on locality analysis for specific cache organizations [20], but not on the data reuse of the program. Essentially this means that a specific analysis must be carried out for each specific data cache to test.

In this article, we propose a general method to obtain safe data reuse information from a binary, independently of the cache levels and data cache characteristics. Essentially, we track the content of registers and memory in each part of the program by means of Abstract Interpretation [7]. We use polyhedra to obtain linear access patterns of data accesses, suitable to be analyzed by means of the well known Loop Nest Data Reuse Theory [33]. This theory provides the mathematical procedures to extract (safe) reuse information between memory instructions.

Although modern compilers perform source-code/intermediate-code data access pattern analyses, they are not suitable for real-time systems. In a real-time context, any analysis must be performed on the final stages or after the compilation in order to take into account the possible code transformations due to optimizations or architectural features (e.g., array padding, vectorization, predicated instructions, etc.). Also, a *safe* analysis is required, whereas compilers perform analyses for the average case. Furthermore, analysis of library calls is only possible by working directly on a statically linked binary, as we propose. Existing WCET frameworks do not perform a deep access pattern analysis. For example, Heptane [13] and AiT [9] carry out an address range analysis for each memory instruction, but they do not provide its specific access pattern.

With our proposal, the reuse properties of each static load/store instruction in the program are detected, independently of the data cache. Essentially, this means that each load/store is linked to the previous load/store accessing the same data (if any), and the reuse type they present. The reuse type will determine the potential always-hit/always-miss/first-hit/first-miss classical categorizations, plus others much more detailed (e.g., 1 miss out of each 8 accesses). Then, a further analysis for a specific data cache can be carried out to confirm these potential categorizations, i.e., for the selected data cache, test whether each reusable cached line is not evicted before it is referenced again.

Our approach has the following strengths:

- It is safe due to the correct use of Abstract Interpretation, meaning that our method only includes situations of guaranteed data reuse.
- Analysis is performed on binary code, enabling the analysis of binaries generated by different compilers and optimization levels.
- The separation between the reuse analysis (as a property of the binary code) and the hit/miss analysis (as the exploitation of such reuse on a particular cache) enables a much more efficient WCET analysis, since our reuse analysis needs to be performed just once, and then apply the detected reuse to as many memory architectures as desired.
- Analysis is completely automatic and needs no manual tuning.

As a proof of concept demonstrator, we have implemented our proposal (available at https://webdiis.unizar.es/gaz/repositories/polygaz using *angr* [25] and *apron* [16], and apply it to TACLeBench [8], considering different compiler optimization levels.

The rest of this article is organized as follows. Related work is outlined in Section II. Section III details the core of our proposal, which extracts the data address generation of each load/store in the program as a linear function, if possible. Section IV shows how to perform a reuse analysis to previous linear functions under the well known loop nest data reuse theory. Our experimentation environment and results are described in Section V. Finally, Section VI presents our conclusions.

II. RELATED WORK

To the best of our knowledge, all WCET analyses of systems with data caches have focused on locality analysis for specific cache organizations [20], but not on the data reuse of the program. Essentially this means that a specific analysis must be carried out for each specific data cache to test. For conventional LRU data caches, this may imply exploring the explicit sequences of data accesses (e.g., [19], [31]), but working in such detail would require an exponential analysis time [2]. To avoid such a problem, Cache Miss Equations (CMEs) [11] or *must/may* analysis [10] can be used, but these approaches present problems for non-perfectly nested loops, accesses to unknown addresses, etc., and they require a prior data access pattern analysis. Huynh *et al.* showed that results of previous methods can be improved by a scope-aware analysis regarding the persistence of contents in cache [14]. However, they need a very specific compiler for their prior data access pattern analysis [5]. Further, they considered write-through caches with a no-write-allocate policy to avoid the analysis of copybacks. This simplifies the analysis, but conventional caches usually follow the opposite approach, namely, write-allocate with fetch on write-miss and copyback, which results in fewer memory transfers in general [17]. A recent study on *must/may* analysis also improves its precision, but it does not consider copybacks

either [28]. Moreover, all these previous approaches are based on tracking the specific value of memory addresses, whereas our proposal represents accesses as expressions and abstract relations, so that reuse is marked when it can be asserted that two data references access the same memory address, independently of whether the address is known or unknown. In order to determine hits and misses without knowing the specific memory addresses accessed, a congruence analysis has been proposed [12]. Such analysis uses Abstract Interpretation with symbolic names, and determines whether accesses are mapped to the same cache set/block, so they may obtain hit/miss information even for accesses to unknown memory addresses. However, this analysis still focus on an LRU cache with a particular number of sets and ways, and it is no longer valid if these parameters change. Compared to this study, our proposal asumes a broader scope (whole memory), uses a more precise abstract domain (polyhedra), and provides equal or more precise relations than the symbolic names and relations.

Alternatively to conventional caches, lockable data caches could be used. A locked data cache is much easier to analyze, and its WCET-aware configuration can be included in the WCET analysis method [30], [34]. Still, the dynamism of data accesses severely restrict the effectiveness of locked data caches. Specifically, a locked data cache cannot exploit array traversals in loops. Thus, such data accesses will be always miss, unless the whole data structure is locked in the data cache [29], [32]. CMEs [11] have been used to estimate whether it is worth locking the whole data structure [29]. If locking is desired, extra code is inserted to preload and lock the corresponding data, although this operation has costs and the drawback of evicting a relatively large portion of the cache. Methods specialized in the analysis of lockable data caches focus on either temporal (e.g., [30], [34]) or spatial reuse (e.g., [11], [29], [32]), whereas our proposal provides both temporal and spatial reuse information.

Another alternative would be to use the predictable *Address-Cache Data-Cache* (ACDC) structure, which can be analyzed as easily as a locked data cache while providing a dynamic behavior similar to that of conventional caches [24]. As all previous caches, it also requires *safe* reuse information for its correct configuration, since the estimated hits and misses depend on the detected reuse, and may affect the WCET.

III. AUTOMATIC EXTRACTION OF DATA ADDRESS GENERATION PATTERNS
A. ABSTRACT INTERPRETATION: OVERVIEW
The exact analysis of all possible memory access patterns of a program is impractical. However, relevant information can be extracted when doing the analysis at a higher level of abstraction. In this work we resort to Abstract Interpretation [7], [10], [12] to obtain reuse information from the memory access patterns generated by the load/store instructions.

Abstract Interpretation is based on a Galois connection between a concrete domain and an abstract domain.

The abstract domain represents over-approximations of subsets in the concrete domain. In our particular framework, the concrete domain is defined by the set of vectors $(r_0, \ldots, r_{n-1}) \in \mathbb{W}^n$ representing the state of a program we are interested in, i.e., all possible values of an integer register file with n registers.[1] The set $\mathbb{W} = \{0, \ldots, 2^w - 1\}$ represents all possible values of a register with a word size of w bits.

The abstract state of a program is represented as a set of invariants that hold for the set of registers. In our case, we use convex polyhedra to represent the elements of the abstract domain as a set of linear inequalities of the form[2]

$$\sum_{i \in \{0, \ldots, n-1\}} c_i r_i \leq k, \qquad c_i, k \in \mathbb{Z}.$$

For example (Figure 1), let us consider a register file with two registers. At a certain point of a program, the registers can hold the values in set S (concrete states represented by solid dots). The abstract state (shadowed area) represents an over-approximation of S, e.g., the state $(1, 1)$ also meets the three invariants but does not belong to S.

Concrete states: (r_0, r_1)
$S = \{(0,0), (2,2), (2,3), (4,2)\}$

Abstract state
$r_0 - 2r_1 \leq 0$
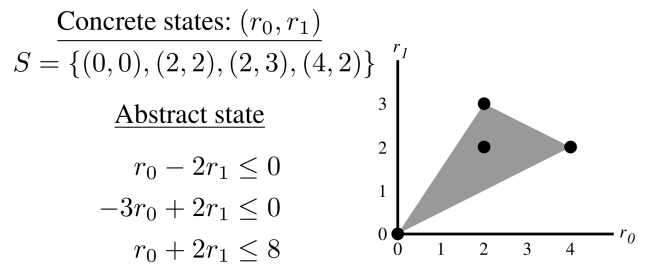$-3r_0 + 2r_1 \leq 0$
$r_0 + 2r_1 \leq 8$

FIGURE 1. Concrete and abstract domains in Abstract Interpretation.

Abstract interpretation guarantees that any safety property holding in the abstract domain also holds in the concrete domain. In particular, it will be used to find safe approximations of data access patterns by analyzing the contents of the registers represented by the abstract states.

An Abstract Interpretation engine computes the abstract states at each point of the program by iteratively visiting the instructions in program order and updating the abstract states until an equilibrium is reached. The computation starts with all abstract states at $\bot$ (empty). They grow until a least fixed-point is reached that represents an over-approximation of the concrete states.

The Abstract Interpretation engine requires a set of functions to transform the abstract states during the traversal of the program. The *transfer* function captures the semantics of each instruction and transforms the abstract state before the execution of the instruction into the abstract state after its execution. An example is shown in Figure 2.

The control flow between basic blocks is captured by the *meet* (union, $\sqcap$) and *join* (intersection, $\sqcup$) functions.

---
[1]For simplicity, in the general description of our proposal we disregard the contents of the memory space, but Section III-E describes how they have been included.

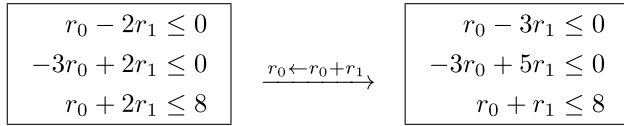[2]Equality constraints can be defined by combining two inequalities.

$$\boxed{\begin{array}{l} r_0 - 2r_1 \leq 0 \\ -3r_0 + 2r_1 \leq 0 \\ r_0 + 2r_1 \leq 8 \end{array}} \xrightarrow{r_0 \leftarrow r_0 + r_1} \boxed{\begin{array}{l} r_0 - 3r_1 \leq 0 \\ -3r_0 + 5r_1 \leq 0 \\ r_0 + r_1 \leq 8 \end{array}}$$

FIGURE 2. Transfer function for the instruction `add r0,r0,r1`.

Finally, *widening* (∇) is a special meet function applied at the back-edges[3] of the loops to guarantee convergence towards a fixed-point. We refer the reader to the theory of Abstract Interpretation [7] for a more detailed discussion on the calculation of the abstract states.

Depending on the desired precision, alternative domains can be used for Abstract Interpretation. Section V includes some discussion on these domains.

B. INTUITIVE EXAMPLE

Let us start with the example in Figure 3, which shows a sequential access to the elements of a matrix A within two nested loops. Assuming an `int` takes 4 bytes and the base address for A (row-major order) being `@A`, the address for `A[i][j]` is computed as:

$$addr(\texttt{A[i][j]}) = \texttt{@A} + 200 \cdot i + 4 \cdot j \quad (1)$$

In general, `@A` can be either a constant, in case of a global array allocated in the static region, or a register, in case it is the parameter of a function.

```
int A[100][50];
for (int i=0; i<100; i++)
  for (int j=0; j<50; j++)
    A[i][j]=0;
```

FIGURE 3. Linear data access pattern dependent on the loop induction variables.

A classical execution sequence for the computation of *addr*(`A[i][j]`) could be as follows:

```
r4 = 200*r2; r2 stores i
r5 = 4*r3; r3 stores j
r6 = r4+r5;
r7 = r1+r6; r1 stores @A, r7=addr(A[i][j])
r8 = load r7; memory access
```

and the following linear invariant could be inferred from the abstract state after the execution of the previous code:

$$r_7 = r_1 + 200 \cdot r_2 + 4 \cdot r_3$$

where r_1 would hold the base memory address, and r_2 and r_3 would hold the values of i and j, respectively.

However, the interesting invariants for memory reuse analysis are often implicit and "hidden" in the representation of complex abstract states, e.g., they must be obtained by computing a linear combination of the explicit invariants of the representation. On the one hand, code optimizations performed by the compiler may transform the relations between

[3]Given two basic block nodes a, b from a control-flow graph, a back-edge is an edge $a \rightarrow b$ whose head b dominates its tail a [1]. All edges that enter the loop header b from the loop body are back-edges.

registers, for instance by using the base address of a data structure as the starting point of an induction variable, or by increasing induction variables using steps different from those in the source code. On the other hand, data held in registers are not only used for address calculation, thus obfuscating the implicit relations required for memory addressing.

An example is shown in Figure 4 with an intermediate representation of the code in Figure 3. The basic blocks are represented by boxes in the control-flow graph. The variables `t1`, `t2` and `t3` are temporary registers used to perform arithmetic operations for address calculation. Figure 4(a) shows the representation after code generation, whereas Figure 4(b) shows the final code after optimization. In this case, the variables `i`, `j`, `t1` and `t2` are identified as induction variables. After strength reduction, multiplications are replaced by additions. The variables `i` and `j` disappear after dead-code elimination under the assumption they are dead after the loops.
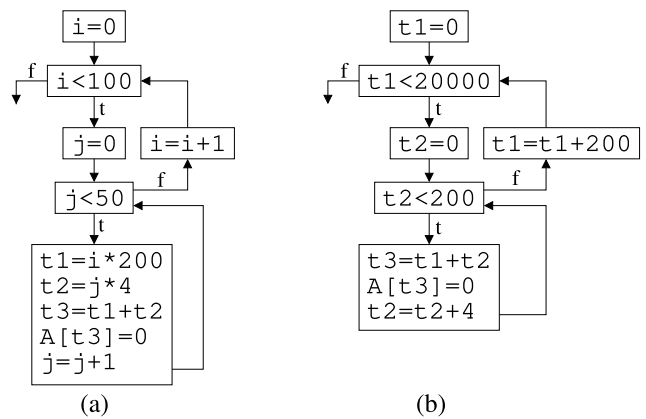

FIGURE 4. Induction variables and strength reduction.

Essentially, our proposal reconstructs the linear access patterns, e.g., eq. (1), from the abstract states, and analyzes the reuse of the memory space.

C. SCOPE OF APPLICATION

Our proposal can be applied to any program that does not contain *non-natural* loops. Natural loops are those with a single-entry node (header) dominating all nodes in the loop, and with at least one back-edge going to the header [1]. Given two natural loops, they are either disjoint or nested.

Roughly speaking, all loops derived from high-level statements (while, for, etc.) are natural. Non-natural loops may appear only in the rare case of using *goto* statements.

D. TRANSFER FUNCTION

The transfer function is associated to the execution semantics of every instruction and defines the relation between the abstract states before and after the execution of one instruction.

We assume that instructions are composed of a sequence of *basic microoperations*, each of them containing no more than a single destination register and a unary/binary

arithmetic/logic operation, e.g., $r_i \leftarrow r_j + r_k$. This microcoding feature is present in tools such as angr [25] and OTAWA [4].

Let us assume that s_{in} and s_{out} are the states before and after executing a micro-operation I, respectively, and r_{dst} is the destination register of I. The transfer function for I modifies r_{dst} and maintains the values of all the remaining registers. It is implemented by three steps:

- $s' \leftarrow s_{in} \cap \{\tau = \text{Transfer}(I)\}$.
- $s'' \leftarrow s' \backslash r_{dst}$.
- $s_{out} \leftarrow s''[r_{dst} := \tau]$.

The first step creates a new state s' with a fresh variable τ and a new relation defined according to the semantics of I. For example, if I is `sub r3,r2,r5` then the constraint $\tau = r_2 - r_5$ is added. The second step eliminates the variable r_{dst} from s', typically using a Fourier-Motzkin elimination [15] for linear inequalities. Finally, the third step substitutes τ by r_{dst} in s''.

For any instruction I of the form $r_{dst} \leftarrow rhs$ (right-hand side), the transfer function is defined as follows:

$$\text{Transfer}(I) = \begin{cases} rhs & \text{if } rhs \in \{r_i, k, r_i \pm r_j, r_i \pm k, r_i \cdot k\} \\ \top & \text{otherwise} \end{cases}$$

where r_i and r_j represent source registers of the instruction and k represents an integer constant. Notice that the transfer function defines the relation $\tau = \top$ when the instruction does not represent any linear relation between registers, e.g., $r_{dst} \leftarrow r_1 \cdot r_2$. The symbol $\top$ represents the top value of the abstract domain.

E. EXTENSION TO TRACK MEMORY CONTENT

Previous scheme performs a value tracking on registers, which works well for most optimized codes. However, non-optimized codes regularly perform register spilling and reloading, which requires extending our tracking to memory. We consider the memory as a large byte-sized register bank, where each memory address is equivalent to a register name, and the access type (byte, word, double word) determines a set of consecutive "registers". With such a view, load/store instructions are equivalent to `mov` instructions, with the following two conservative exceptions: (1) a load from an unknown address assigns $\top$ to the destination register, and (2) a store to an unknown address assigns $\top$ to *all memory positions*.

F. INDUCTION VARIABLES

Induction variables are those increased or decreased by a fixed amount at each iteration of a loop, and play an essential role in array indexing within loops. Some induction variables depend linearly on other induction variables [1].

1) DETECTION OF INDUCTION VARIABLES

We exploit the use of Abstract Interpretation for the detection of induction variables stored in registers within natural loops. The following scheme is used. For every candidate r, a new variable r^{prev} is created to store the value of r at the previous iteration. At the entrance of the loop header, the assignment $r^{prev} = r$ is introduced. At every back edge of the loop the assignment $r^{step} = r - r^{prev}$ is evaluated.

An induction variable is detected when the union of the states that traverse all back edges of the loop fulfill that:

$$r^{step} = k, \text{ for some } k \in \mathbb{Z}.$$

Notice that this scheme is more general than the classical structural methods for detecting of induction variables. It also detects those variables with multiple assignments within the loop that result in an constant accumulated value after all the assignments, even in the presence of conditional statements.

2) ASSOCIATION OF LOAD/STORE ADDRESSES WITH INDUCTION VARIABLES

Target memory addresses generated by load/store instructions may be either constant (access to scalars), linear with respect to an induction variable (sequential access), or non-linear. Depending on whether the analyzed CFG represents each function just once or once per call, stack accesses (relative to the stack or frame pointer) will be non-linear or constant, respectively.

During the Abstract Interpretation analysis, both the abstract state at the load/store program point, and the variable name (virtual or actual processor register) associated to the target memory address are known.

For accesses to constant addresses, obtaining such value is immediate by extracting its interval of values from the target variable, and verifying that both upper and lower bounds are constants and coincide.

For sequential accesses to arrays in a single loop, linear expressions usually follow this scheme:

$$targetAddr = baseAddr + step \cdot inductionVar$$

After detecting the induction variables and performing strength reduction (as shown in the example of Figure 4), the address calculation is reduced to a scheme like this:

$$targetAddr = targetAddr + step$$

where $targetAddr$ is initialized to $baseAddr$ before entering the loop and $step$ is a known constant at compile time.

In nested loops accessing multidimensional arrays, a different induction variable is typically used at each level. After applying the optimizations for strength reduction, the resulting calculation has the following scheme for n nested loops:

$$targetAddr = baseAddr + \sum_{i=1}^{n} t_i$$

where t_i represents the induction variables (usually in registers) that store the corresponding offsets. These variables are updated by increasing/decreasing their values by an integer constant. In Figure 4, the variables `t1` and `t2` play the role of induction variables for the two dimensions of the array `A`.

If constant values for *baseAddr* and all t_i steps can be inferred, a precise linear access pattern such as the one in eq. (1) is generated. If steps are known but the *baseAddr* value cannot be inferred, the generated pattern will be imprecise, which will prevent group-reuse detection, as described below. In any other case, the memory reference is considered non-linear.

IV. DATA REUSE

In this section we describe the detection of data reuse based on the information gathered from previous analysis. Such information can be either a precise linear access pattern (constant accesses, or sequential accesses with a known stride and base address), a linear access pattern (known stride) with an unknown base address, or a non-linear access pattern.

A. DATA REUSE FOR PRECISE LINEAR ACCESS PATTERNS

Our reuse analysis is based on loop nest data reuse theory [33], briefly introduced below. Each iteration in a loop nest corresponds to a node in the *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 $f(\vec{i}) = \vec{h} \cdot \vec{i} + c$, where matrix H has been transformed into a vector $\vec{h}$ and vector $\vec{c}$ has been transformed into a constant c. So, $c = baseAddr$ and $\vec{h}$ is composed of the corresponding steps t_i.

For each memory instruction with a linear access pattern function, the next step is to recognize reuse among accesses of the same memory instruction (*self reuse*) or among accesses of different memory instructions (*group reuse*). As above, let us outline the mathematical procedures provided by the loop nest data reuse theory to detect such reuse [33].

Essentially, group reuse appears for two distinct references $H_1\vec{i} + \vec{c}_1$ and $H_2\vec{i} + \vec{c}_2$ if $H_1 = H_2$ and $\vec{c}_1 = \vec{c}_2$. Although other group reuse situations could easily be detected (e.g., stencil codes such as "`for (i=0;i<100;i++) A[i]=A[i+1];`"), in this article we consider just identical patterns, since it is the most common group reuse case. Hence, our process for group reuse detection for a given load/store consists of testing whether there is another load/store with the same access pattern with a dominance relation (i.e., it always appears earlier in *program order*).

Regarding self reuse, it is classified as *self-temporal*, when the same data element is repeatedly accessed in time (e.g., accessing repeatedly a scalar variable in a loop), or *self-spatial*, when close elements are accessed following a particular pattern (e.g., traversing an array in a loop). Self-temporal

reuse happens when a reference $H\vec{i} + \vec{c}$ accesses the same data element in iteration $\vec{i}_1$ and $\vec{i}_2$, that is, $H\vec{i}_1 + \vec{c} = H\vec{i}_2 + \vec{c}$. The solution of this equation can be obtained by applying the kernel operation on H [33]. Self-spatial reuse can be detected in the same way, but using a truncated H, with all elements of its last row replaced by 0.

In the presence of caches, reuse arises among instances of memory references if they target to the same data memory line, not only to the same byte, word, or double word. So, *temporal reuse* for scalars includes all loads/stores that access an already accessed memory line. This already accessed memory line may be brought from memory by the same load/store that reuses it (classified as *self-temporal reuse*), or may be brought by a previous load/store not reusing it (classified as *first use*). Arrays follow a similar classification. *Short self-spatial reuse* represents array traversals with a stride/step small enough that at least every other consecutive memory access is serviced from the same data cache line, while *long self-spatial reuse* represents array traversals with a stride too long to guarantee hits in general, e.g., traversing an array by columns. *Group-temporal reuse* represents loads/stores with an access pattern function identical to an earlier load/store, with such an earlier load/store classified as either short or long self-spatial reuse. Figure 5 shows examples of these categories.

```
int A[100][100]; int b, c, d;
b=1;                 // 1: scalar first use
for (register int i=0; i<100; i++) {
  c=                 // 2: scalar self-temporal reuse
    A[i][5]+         // 3: array long self-spatial reuse
    b;               // 4: scalar group-temporal reuse, reusing 1
  A[5][i]=           // 5: array short self-spatial reuse
    A[i][5]+         // 6: array group-temporal reuse, reusing 3
    c; }             // 7: scalar group-temporal reuse, reusing 2
```

FIGURE 5. Example of reuse classifications, assuming that scalar variables are located at *different* memory lines and the compiler has not optimized, except for allocating variable *i* to a register.

B. REUSE FOR NON-LINEAR OR IMPRECISE LINEAR ACCESS PATTERNS

For linear access patterns with an unknown base address, we have $f(\vec{i}) = \vec{h} \cdot \vec{i} + c$ where c is unknown. This means that *self-spatial reuse* can be detected, since detection does not require the c value. However, a further hit/miss analysis on such self-spatial reuse cannot provide the specific addresses generating hits/misses, but only the hit/miss ratio. On the other hand, *group reuse* cannot be detected with the data reuse theory, since it requires the c value. Nevertheless, group reuse can be detected from the Abstract Interpretation analysis. If the two variables holding the target data address of the two accesses to compare are identical, they present group reuse, even if the the base address is unknown.

For non-linear access patterns, the data reuse theory is not applicable. Nevertheless, group reuse can be detected from the Abstract Interpretation analysis as above.

Figure 6 shows several examples for these data accesses.

```
int A[100], B[100]; register int * p;
register listElement * p2;
if (cond) { p=A; } else { p=B; }
for (register int i=0; i<100; i++) {
  p[i]=0;          // 1: array short self-spatial, base un-
known
   if (p2->next!=NULL)  // 2: non-linear
     p2=p2->next; // 3: non-linear group-temporal, reusing 2
}
```

FIGURE 6. Example of reuse classifications, for non-linear and imprecise linear memory accesses.

V. RESULTS

In this section we evaluate our safe data reuse detection method. First, we describe the experimental framework and benchmarks. Then, we study the reuse classification of their data memory accesses. With this reuse classification, a simple estimation on how much time they would take in the worst case is computed, i.e. their contribution to the WCET. Finally, we present the analysis times required to apply our method.

A. EXPERIMENTAL FRAMEWORK AND BENCHMARK CHARACTERIZATION

The 33 benchmarks tested have been compiled with the different optimization levels (-O0, -O1, -O2, -O3) with *gcc-9.2.1* for ARM, disabling thumb extensions, as previous studies [22]. We assume that loop bounds (*flow-facts*) are provided in advance, as in other frameworks (e.g., OTAWA [4]). For each one of the binaries, this information has been manually set, carefully studying the effect of compiler optimizations. Nevertheless, existing loop bound analysis methods could be used [6], [18].

Table 1 shows the benchmarks used in our experiments, from the TACLeBench suite [8]. Recursion has not been addressed in this work, so recursive benchmarks have been discarded. We use *angr* (version 8.19.7.25) to extract and process the CFGs [25]. 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[4] have been discarded. Functions in the CFG are not virtually inlined, that is, in our analysis a function is associated to a unique sub-CFG, and not associated to a specific sub-CFG instance for each call it receives. So, we first process each node in the CFG following a fixed-point algorithm, generating an abstract state at each load/store instruction and the input/output abstract states for each node. For the Abstract Interpretation analysis, we use *apron* (version 0.9.10) with the polka (equalities mode) polyhedra library [16]. That is, each update in a register or memory address is translated to a call to the *apron* library updating the specified data. Once the CFG is completely processed, we perform another Abstract Interpretation analysis to discover the registers holding induction variables in loops. Results of both analyses are then combined

to generate access patterns. Finally, knowing the access pattern for each load/store in the CFG, the reuse information is generated. Figure 7 depicts this process as a flowchart.
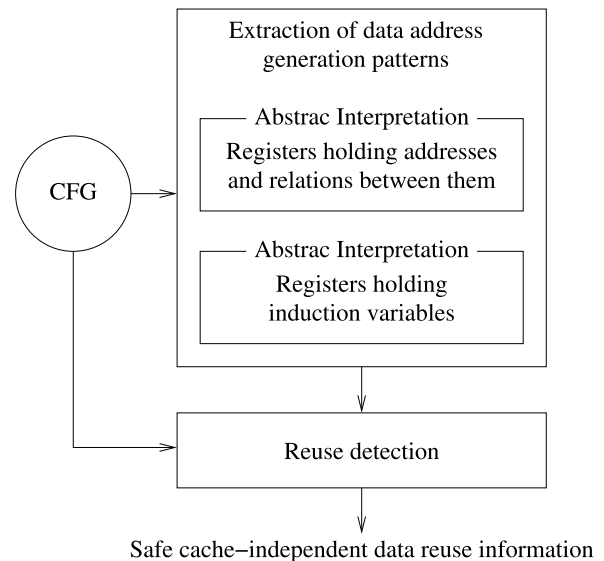

FIGURE 7. Flowchart for obtaining load/store access patterns and reuse information between them.

In Table 1, for each benchmark and optimization level, columns under "Dominant loads/stores with exploitable reuse" show the number of load/store instructions that bring content with exploitable reuse from memory, for the first time (i.e., with a dominant relation). They show both the absolute value, and the percentage out of the total number of static loads/stores for each optimization level (O0 to O3). O0 usually contains unnecessary temporal variables that optimizations remove, and hence column O0 in general presents more dominant loads/stores than the other columns. In order to provide a more realistic insight, accesses to scalars are presented considering a cache line of 64 bytes (typical for Intel and ARM L1 caches). So, a scalar access is assumed to cache a whole memory line, and not only its accessed bytes. Thus, all results in this section regarding scalars appear as if they were grouped by their memory location in blocks of 64 bytes. Columns under "Estimated accesses in the worst case" show an estimation of the number of dynamic (accounting loop iterations) data memory accesses for each binary in the worst case, as an absolute value for O0, and the percentage in respect of O0 for optimized binaries. As can be seen, optimizations markedly reduce the number of memory accesses. Finally, columns under "Analysis time (s)" show the time required to complete the analysis, in seconds, in a 3.20 GHz Intel Core i5-4570 CPU with 16 GiB of RAM. Benchmark *test3* requires more virtual memory when compiled with O0 (134 GiB) and O1 (37 GiB). This benchmark overwrites the same global variables many times from functions, but optimization levels 2 and 3 remove most of them, reducing very much the total number of memory accesses and the required analysis time. For these binaries, and also for *mpeg2*

---

[4]adpcm_dec, adpcm_enc, ammunition, cjpeg_transupp, cjpeg_wrbmp, cover, duff, epic, gsm_dec, gsm_enc, h264_dec, ndes, prime, petrinet, sha, susan, and also for bsort-O1, lift-O2, and lift-O3.

TABLE 1. Benchmark characterization.

Name	Dominant loads/stores with exploitable reuse				Estimated accesses in the worst case				Analysis time (s)			
	O0	*O1*	*O2*	*O3*	*O0*	*O1*	*O2*	*O3*	*O0*	*O1*	*O2*	*O3*
audiobeam	173 (18%)	45 (12%)	46 (12%)	24 (8%)	603 746	14%	12%	9%	1134	1124	742	40 345
binarysearch	20 (21%)	7 (13%)	5 (16%)	6 (4%)	703	43%	21%	21%	3	2	1	93
bsort	3 (3%)	—	2 (7%)	1 (5%)	257 237	—	15%	15%	1	—	1	1
complex_updates	20 (15%)	7 (10%)	6 (9%)	16 (8%)	1206	23%	22%	16%	13	7	7	377
cosf	38 (16%)	9 (15%)	9 (16%)	6 (16%)	26 825	14%	12%	6%	49	24	29	47
countnegative	24 (21%)	12 (16%)	8 (13%)	8 (13%)	11 685	31%	17%	17%	8	3	2	2
cubic	164 (17%)	34 (11%)	33 (12%)	32 (12%)	4 864 948	4%	3%	3%	56 515	12 886	11 086	15 396
deg2rad	6 (14%)	6 (24%)	2 (22%)	2 (22%)	5081	0%	0%	0%	1	0	0	0
dijkstra	33 (15%)	15 (10%)	11 (7%)	14 (9%)	2 999 224 676	21%	20%	19%	36	54	162	72
fft	50 (23%)	20 (15%)	18 (16%)	16 (16%)	39 583 818	66%	66%	66%	26	62	61	49
filterbank	3 (5%)	3 (11%)	3 (9%)	3 (6%)	6556	12%	12%	12%	1	1	2	10
fir2dim	10 (6%)	5 (6%)	3 (4%)	11 (7%)	3871	15%	12%	11%	63	19	12	192
fmref	108 (17%)	33 (12%)	31 (13%)	40 (13%)	843 457	13%	12%	24%	5753	1345	1303	12 663
g723_enc	104 (14%)	36 (13%)	54 (19%)	40 (12%)	921 750	26%	14%	8%	3250	1192	4026	3205
huff_dec	55 (20%)	13 (9%)	16 (12%)	20 (9%)	470 226	37%	30%	19%	909	369	414	1147
iir	7 (7%)	8 (19%)	7 (16%)	9 (17%)	1238	14%	14%	14%	6	2	3	5
insertsort	22 (13%)	12 (9%)	12 (10%)	13 (8%)	3262	13%	13%	15%	39	41	29	29
isqrt	22 (18%)	10 (17%)	7 (14%)	7 (11%)	696 715	2%	2%	4%	7	5	4	19
jfdctint	10 (3%)	20 (22%)	9 (8%)	9 (8%)	3551	14%	19%	19%	149	47	55	56
lift	73 (12%)	37 (9%)	—	—	1 067 341	28%	—	—	2689	855	—	—
lms	34 (17%)	8 (10%)	8 (10%)	7 (10%)	138 252	10%	10%	9%	146	26	45	38
ludcmp	38 (12%)	11 (9%)	12 (11%)	48 (14%)	7519	13%	12%	16%	28	49	49	344
matrix1	7 (7%)	6 (11%)	5 (11%)	6 (7%)	7271	38%	37%	29%	9	5	3	17
md5	66 (5%)	29 (11%)	36 (13%)	61 (13%)	41 892 631	13%	9%	11%	10 304	8183	7650	53 820
minver	52 (13%)	19 (10%)	17 (11%)	37 (11%)	3511	22%	17%	18%	66	200	134	1298
mpeg2	457 (14%)	164 (7%)	185 (8%)	254 (10%)	88 196 840 655	1%	1%	0%	16 872	203 668	128 748	1 301 669
pm	127 (12%)	23 (7%)	23 (6%)	27 (7%)	8 572 824	8%	11%	27%	27 180	29 104	36 783	60 144
powerwindow	256 (15%)	99 (10%)	70 (8%)	80 (8%)	3 039 783	50%	44%	41%	23 712	4653	17 173	15 727
rad2deg	7 (17%)	6 (25%)	2 (25%)	2 (25%)	5067	0%	0%	0%	1	0	0	0
rijndael_dec	102 (9%)	66 (12%)	86 (12%)	70 (10%)	2 269 935	38%	47%	49%	5073	843	1432	1517
rijndael_enc	139 (8%)	70 (12%)	101 (14%)	93 (11%)	2 930 110	28%	33%	34%	14 606	1009	1632	2553
st	42 (19%)	15 (13%)	7 (9%)	11 (13%)	85 359	21%	9%	9%	191	27	15	35
test3	772 (4%)	561 (13%)	12 (8%)	34 (16%)	394 139 505	12%	0%	0%	559 595	156 770	28	90

compiled with O3 (77 GiB), a different machine has been used. It is important to take into account that this analysis must be seen as a proof of concept, and currently it is not optimized. Nevertheless, notice that our analysis should be conducted just once for each binary in order to extract its reuse properties, independently of the data cache. Then, for each specific data cache, a much more simple analysis on top of our reuse analysis should provide the detailed hit/miss cases. That is, test which reuse cases, from those provided by our analysis, can be effectively exploited by the selected data cache.

Figure 8 shows the number of dominant loads/stores with exploitable reuse of Table 1, with benchmark ordered by their value for O0. Such ordering provides an insight into the data complexity of each benchmark regarding their reusable data. As it can be seen, similar trends appear when applying optimizations, although in general with lower absolute values. All figures in this section follow this ordering for the benchmarks.

Despite the different absolute number of dominant references among binaries, the percentage of dominant references out of total references is rather homogeneous (around 12.5%).

B. REUSE CLASSIFICATION

In Figure 9, we present our reuse classification, as described in Section IV, weighting each load/store by its maximum number of executions in loops. Benchmarks are ordered as

above to provide an insight of their data complexity. The different reuse types are separated by their associated data access type (scalar, array, non-linear). The average case is shown on the right. For each data access type, bar fill color shows the specific reuse type detected. Types with darker colors are prone to generate always hits, whereas lighter colors would usually turn into misses.

Figure 9 shows that the compiler optimization level may significantly change the amount and relative distribution among reuse types. A plain compilation (O0) usually has many temporal variables and replicated memory accesses, as seen in Table 1, this increasing group-temporal reuse on scalars. Optimizations remove these unnecessary accesses, and hence the scalar bar is reduced with optimizations. This can be seen in many benchmarks, specially in those on the left.

Compiling without optimizations also hinders the detection of array accesses, which many times are classified as non-linear accesses. With optimizations, sequential accesses are easier to detect, as can be seen, for instance, in *isqrt*, *countnegative*, and *ludcmp*. This is also reflected in the average.

The full unrolling of loops with memory instructions also affects reuse. When compiling with O3, the short stride self-spatial reuse (array) in the affected loops disappears in favor of group-temporal reuse (scalar). This can be clearly seen in *complex_updates*, and in a lesser extent also in
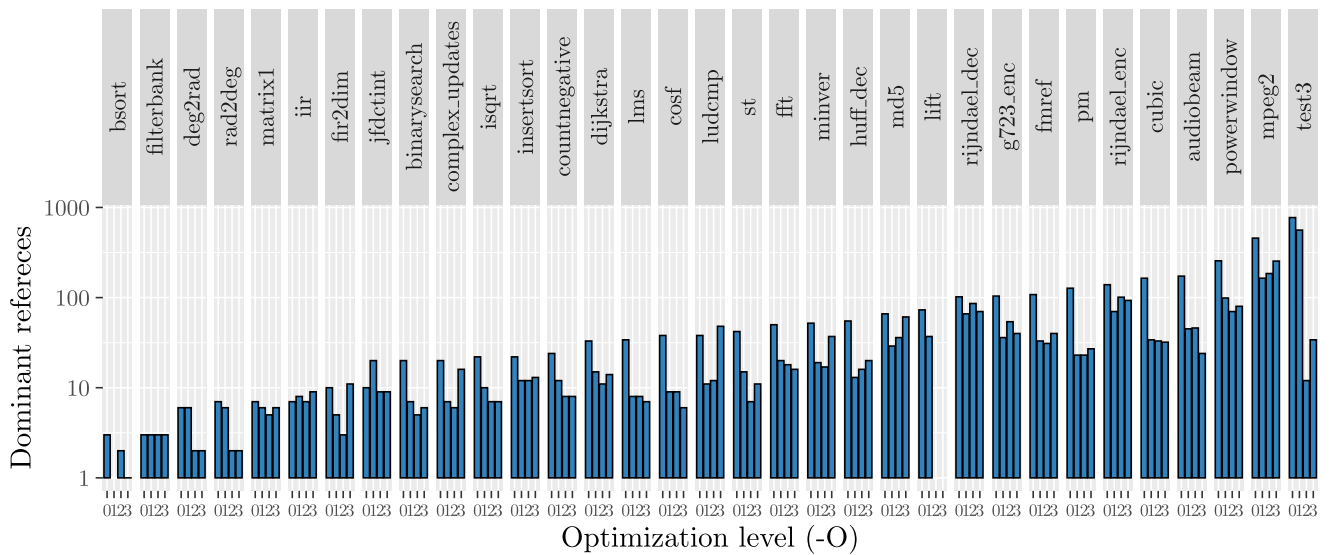
FIGURE 8. Absolute number of dominant load/store instructions with exploitable reuse.

matrix1, *binarysearch*, *ludcmp*, *minver*, and *lift* comparing to O2. Unrolling can also produce other effects regarding locality exploitation. For instance, *filterbank* processes two 8×32 arrays by columns, and when applying O3 the deepest loop is fully unrolled. So, this loop is substituted by a sequence of 8 instructions for each array which individually access the arrays by rows. Thereby, long self-spatial reuse is transformed into short self-spatial reuse. Although short self-spatial reuse is much better for the WCET, notice that the opposite transformation would occur if these nested loops were interchanged.

Regarding "Not reusing (scalar)" cases, in general they represent a very small fraction of the memory accesses, with *deg2rad* and *rad2deg* in O1, O2, and O3 being the most clear exceptions. As it can be seen in Table 1, the number of memory accesses when optimizing these benchmarks drops to less than 0.5%, since 17 memory instructions in *deg2rad* and 18 in *rad2deg* are removed from the only loop in each benchmark. So, with optimizations O1, O2, and O3, the number of memory accesses is 25, 9, and 9 for *deg2rad*, and 24, 8, and 8 for *rad2deg*, respectively. Thus, the first time they access each scalar variable, accounted as "Not reusing (scalar)", is a significant percentage out of the few performed accesses.

C. INTEGRATION IN THE WCET ANALYSIS

The methodology described above enables the safe data reuse detection from data access patterns of load and store instructions. Such reuse information is valid for any data cache hardware, since reuse is a property of the compiled code. Depending on the data cache, such reuse may or may not be exploited as cache hits. In general, this would require an additional safe data interference/interleaving analysis on top of our safe data reuse information, i.e., a persistence analysis [14], although it should be straightforward:

- Memory instructions tagged as "not reusing" (scalar and non-linear) should be accounted as unknown (misses, if no timing anomalies are considered) in the worst case.
- Loads/stores classified as "group-temporal" reuse (scalar, array, and non-linear) would always hit, unless the accessed content is evicted between reusing accesses. Although such analysis is beyond the scope of this article, it could be addressed by comparing the the *minimal life-span* [23] of the target data cache to the number of loads/stores between two data accesses with reuse.
- Scalar accesses classified as "self-temporal" should generate an unknown (miss, without timing anomalies) for its first access in the loop, and hits for all other accesses in the loop (assuming it is not evicted between accesses).
- Array accesses classified as "short self-spatial" reuse (e.g., a large array traversal accessing 8 elements per cache line, sequentially) should generate a significant hit ratio, depending on the element size, cache line size, stride, and base address, again verifying that the accessed content is not evicted between accesses.
- Array accesses classified as "long self-spatial" reuse (e.g., an array of large structs where a single small field per struct is accessed) should be considered as unknown (always miss, if no timing anomalies are considered) in the worst case (unless the cache is able to hold the whole data structure, which would generate a hit ratio similar to accesses classified as "short self-spatial" reuse).

In order to provide some insight into the application of our proposal, we perform a simple rough estimation on the execution time devoted to data accesses. For each binary, we assume that *all* load/store instructions are executed, each one weighted by the maximum number of times
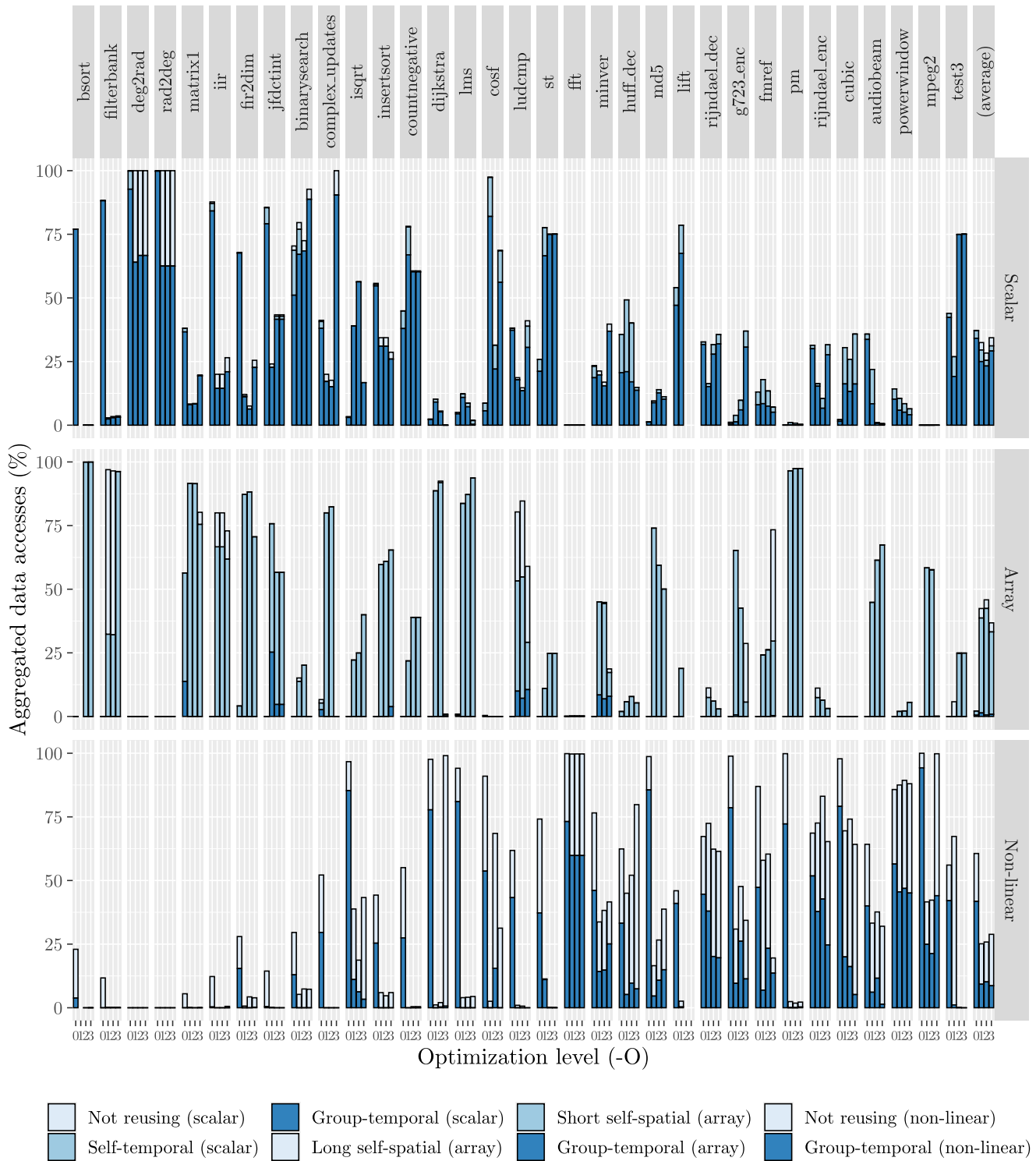
FIGURE 9. Percentage of reuse types found in each binary. Each load/store operation in the binary is weighted by its maximum possible number of executions.

derived from the loop bounds. We count as misses those accesses from instructions classified as "not reusing" and "long self-spatial" reuse, and as hits those accesses from instructions classified as "group-temporal". We consider the

"self-temporal (scalar)" reuse cases as one first miss, and hits for the rest of the accesses in the loop. For the "short self-spatial" reuse (array sequential accesses), we assume a cache line able to hold 8 elements, resulting in one miss
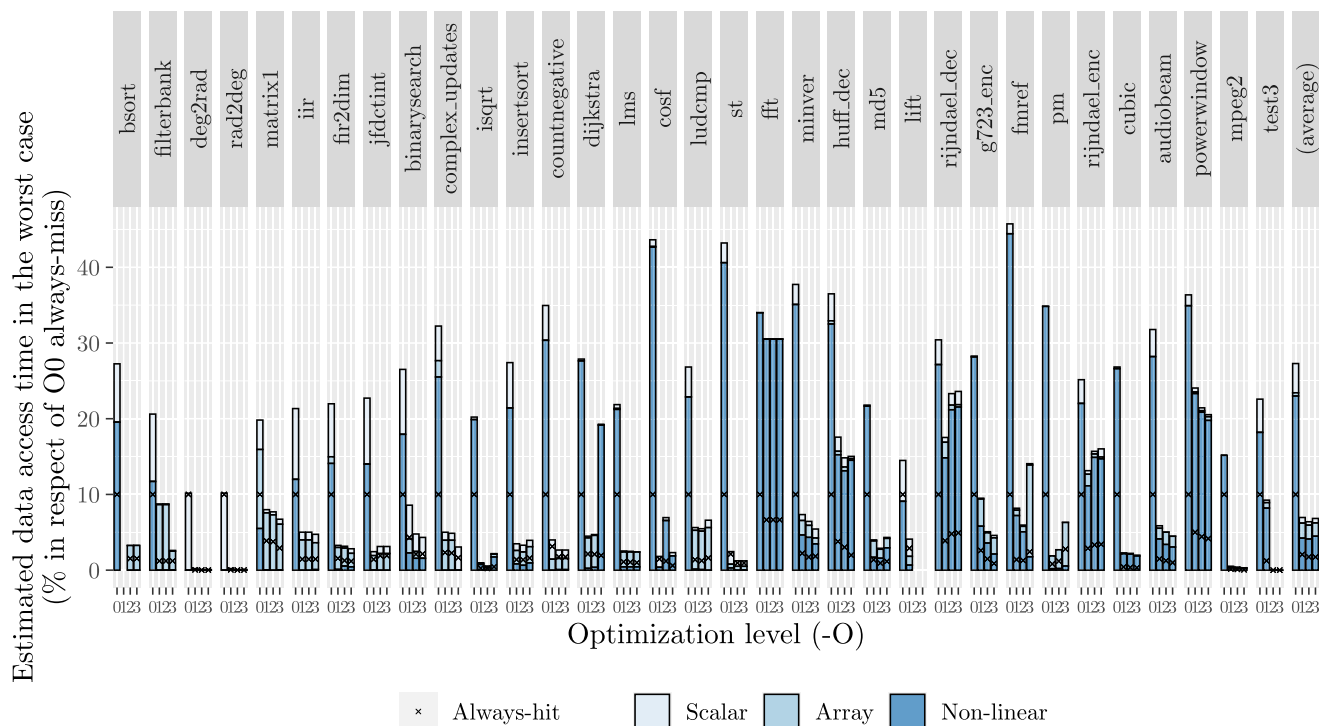
FIGURE 10. Rough estimation of worst case execution time devoted to data memory accesses in respect of an always-miss O0 binary.

out of 8 accesses. This would be equivalent to accessing elements of type *double* with cache lines of 64 B (typical Intel and ARM L1 caches). All these assumptions are reasonable (in general more pessimistic than optimistic) for a general application running on a system with just a small L1 data cache.

Assuming the estimation of hits and misses described above, Figure 10 shows the time devoted to data accesses in the worst case, in respect of the always miss case for the O0 binary. We consider a hit cost of a single cycle, and a miss cost of 10 cycles. Higher values would improve the benefits of a good reuse detection. Execution time of instructions is not accounted. Hence, bars tagged as O0 show the execution time reduction due to the estimated hits, and the remaining bars also consider the reduction of the number of data accesses due to optimizations. For each column, the mark × shows the (unreachable) always-hit bound. With previous hit/miss costs, for O0 the always-hit bound is 10% of the always-miss by construction.

As it can be seen, the resulting times in Figure 10 represent a very low percentage. Focusing on the results without optimizations (O0 columns), worst-case times represent 27% in average in respect of the always miss case. Considering the previous hit/miss times, this corresponds to a data hit ratio of 81%, with misses generally associated to non-linear data accesses. Both *deg2rad* and *rad2deg* are exceptions, since they present a data hit ratio of 99.7%. With optimizations (columns 1, 2, 3) the time required for data accesses drops

to 6.5% in average. By optimizing the code, unnecessary data memory accesses are removed, so in general there are far less accesses, as already seen in Table 1. Also, the percentage of accesses classified as non-linear is usually lower with optimizations, since both data memory addressing and address computations are simplified in the process.

D. ANALYSIS TIME
In this section we study the analysis time required to apply our method, both with polyhedra and octagon domains. It must be taken into account that our analysis has been implemented as a proof of concept, and currently it is not optimized. We use *apron* [16] as our Abstract Interpretation engine, which is reported to be slower than others [26]. Also, *angr* [25] runs on *python*, an interpreted language that is usually much slower than standard compiled codes.

Values in all previous figures and tables use the polyhedra domain for the Abstract Interpretation analysis. The polyhedra domain is expressively richer than octagons (relations such as $x_i - x_j \geq k$). So, octagons should generate less precise results (e.g., more non-linear cases), and more data cache misses should be expected in the worst case. However, after performing the equivalent analysis with octagons, the obtained results (not shown) have been identical or with inappreciable differences. Hence, for the tested benchmarks, octagons suffice for our methodology.

Figure 11 shows the analysis times using polyhedra (polka equalities mode, as in Table 1), and also using the *apron*
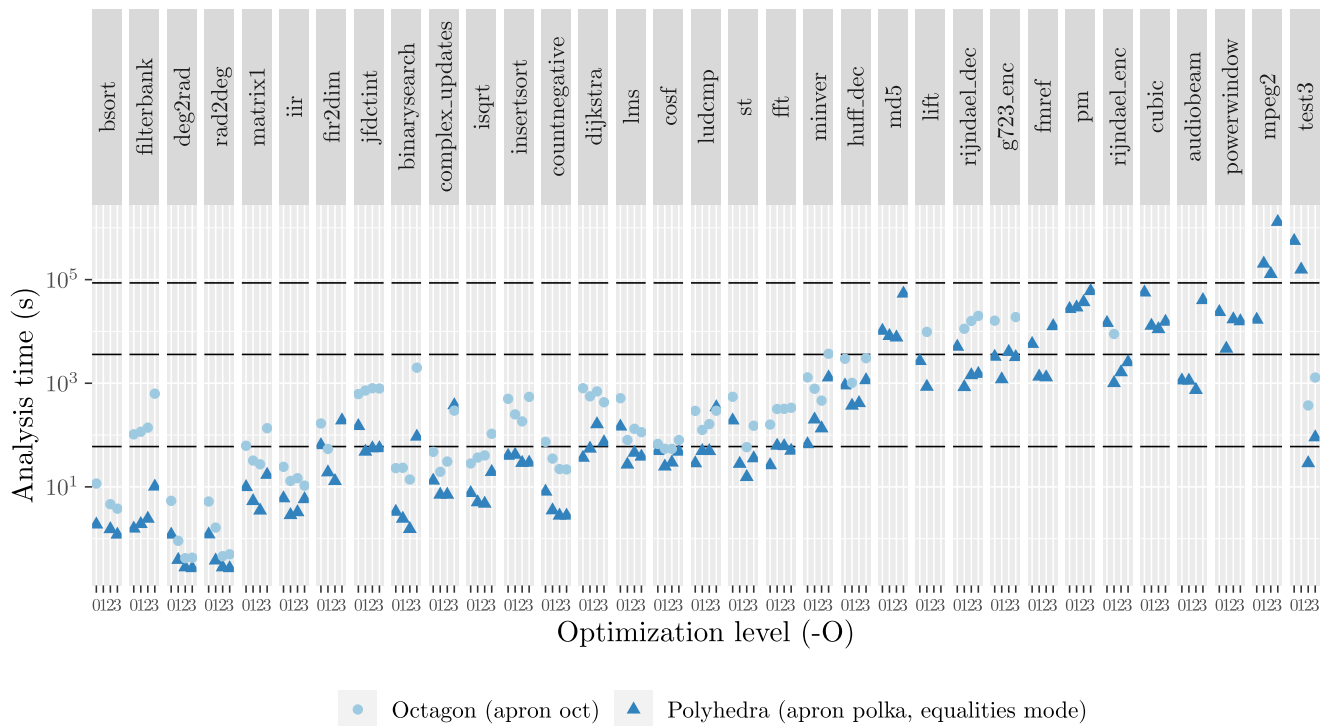
FIGURE 11. Analysis time, with horizontal lines representing 1 minute, 1 hour, and 1 day. Octagon experiments taking more than 1 day are not shown.

octagon domain. Since polyhedra are more complex than octagons, polyhedra should require more computation time, but it is not so for our analysis. The analysis time ratio between octagons and polyhedra has a median over 5. Focusing on polyhedra, O0 and O3 present similar analysis times, whereas O1 and O2 require half the analysis time than O0 and O3, in median. Although performance of numeric analysis is out of our scope, possibly our analysis takes profit of the sparse representation in the polyhedra library. On the other hand, apron octagons use a dense representation, which would be inefficient for our analysis.

VI. CONCLUSION

In this article, we propose a general method to extract safe data access patterns for the load/store instructions in a program, and detect reuse between their accesses. Since data reuse is a property of the program, such information is independent from the data cache of the target system (LRU, locked, ACDC, etc.). Depending on the specific data cache, such reuse may or may not be exploited as cache hits. Thus, our safe reuse information can be used as the basis for the WCET analysis of systems with any data cache. Our proposal analyzes binary code, does not require the exploration of explicit data access sequences, provides both temporal and spatial reuse information, manages references to unknown addresses, and is completely automatic. It uses Abstract Interpretation, tracking the operations carried on the registers and memory. Such operations are transformed into relations and

data access pattern functions for the corresponding load/store instructions, and then a reuse analysis is performed.

As a demonstrator, we have implemented our proposal (available at https://webdiis.unizar.es/gaz/repositories/polygaz using *angr* and *apron*. Apart from validating our reuse detection method, we characterize the data accesses of TACLeBench benchmarks for different optimization levels. Further, we calculate a simple estimation on the worst-case time devoted to data accesses for these benchmarks. Without optimizations, our results show that the time devoted to data accesses in the worst case is reduced to 27% compared to an always-miss system. That is, our proposal guarantees a data hit ratio of 81% in the worst case. With optimizations, such time is reduced to 6.5%.

REFERENCES

[1] V. A. Aho, S. M. Lam, R. Sethi, and D. J. Ullman, "Compilers: Principles, techniques, and tools," in *Addison-Wesley Series in Computer Science / World Student Series Edition*. Reading, MA, USA: Addison-Wesley, 2007.

[2] L. C. Aparicio, J. Segarra, C. Rodriguez, J. L. Villarroel, and V. Vinals, "Avoiding the WCET overestimation on LRU instruction cache," in *Proc. 14th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2008, pp. 393–398.

[3] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, "Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems," *J. Syst. Archit.*, vol. 57, no. 7, pp. 695–706, Aug. 2011.

[4] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An open toolbox for adaptive WCET analysis," in *Software Technologies for Embedded and Ubiquitous Systems* (Lecture Notes in Computer Science), vol. 6399, S. L. Min, R. G. Pettit IV, P. P. Puschner, T. Ungerer, Eds. Waidhofen/Ybbs, Austria: Springer, Oct. 2010, pp. 35–46.

[5] M. E. Benitez and J. W. Davidson, "A portable global optimizer and linker," in *Proc. ACM SIGPLAN Conf. Program. Lang. design Implement. - PLDI*, Jun. 1988, pp. 329–338.

[6] A. Bonenfant, M. D. Michiel, and P. Sainrat, "oRange: A tool for static loop bound analysis," in *Proc. Workshop Resource Anal.*, 2008, p. 35–468.

[7] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. POPL*, 1977, pp. 238–252.

[8] 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 *Proc. 16th Int. Workshop Worst-Case Execution Time Anal., WCET*, vol. 55, M. Schoeberl, Ed. Toulouse, France: OASICS, Jul. 2016, pp. 2:1–2:10.

[9] C. Ferdinand and R. Heckmann, "Ait: Worst case execution time prediction by static program analysis," in *Building the Information Society* (IFIP International Federation for Information Processing), vol. 156, R. Jacquart, Ed. Toulouse, France: Springer, Aug. 2004, pp. 377–383.

[10] C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems," *Real-Time Syst.*, vol. 17, nos. 2–3, pp. 131–181, 1999.

[11] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 703–746, Jul. 1999.

[12] S. Hahn and D. Grund, "Relational cache analysis for static timing analysis," in *Proc. 24th Euromicro Conf. Real-Time Syst.*, Jul. 2012, pp. 102–111.

[13] D. Hardy, B. Rouxel, and I. Puaut, "The heptane static worst-case execution time estimation tool," in *Proc. 17th Int. Workshop Worst-Case Execution Time Anal., WCET*, vol. 57, J. Reineke, Ed. Dubrovnik, Croatia: OASICS, Jun. 2017, pp. 8:1–8:12.

[14] B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *Proc. 17th IEEE Real-Time Embedded Technol. Appl. Symp., RTAS*, Chicago, IL, USA, Apr. 2011, pp. 203–212.

[15] J.-L. Imbert, "Fourier's elimination: Which to choose," in *Proc. Int. Conf. Princ. Pract. Constraint Program.*, 1993, pp. 117–129.

[16] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 5643, A. B. O. Maler, Ed., Grenoble, France: Springer, Jun. 2009, pp. 661–667.

[17] N. P. Jouppi, "Cache write policies and performance," in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, May 1993, pp. 191–201.

[18] H. Li, I. Puaut, and E. Rohou, "Traceability of flow information: Reconciling compiler optimizations and WCET estimation," in *Proc. 22nd Int. Conf. Real-Time Netw. Syst. - RTNS*, 2014, p. 97.

[19] Y.-T.-S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *Proc. 17th IEEE Real-Time Syst. Symp.*, Dec. 1996, pp. 254–263.

[20] M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi, "A survey on static cache analysis for real-time systems," *Leibniz Trans. Embedded Syst.*, vol. 3, no. 1, pp. 05:1–05:48, 2016.

[21] T. Mitra, "Time-predictable computing by design: Looking back, looking forward," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, p. 153.

[22] A. Pedro-Zapater, J. Segarra, R. G. Tejero, V. Viñals, and C. Rodríguez, "Reducing the WCET and analysis time of systems with simple lockable instruction caches," *PLoS ONE*, vol. 15, no. 3, pp. 1–21, Mar. 2020.

[23] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, Sep. 2007.

[24] J. Segarra, C. Rodríguez, R. Gran, C. L. Aparicio, and V. V. Nals, "ACDC: Small, predictable and high-performance data cache," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 2, pp. 38:1–38:26, 2015.

[25] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 138–157.

[26] G. Singh, M. Püschel, and T. M. Vechev, "Fast numerical program analysis with reinforcement learning," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 10981, H. C. G. Weissenbacher, Ed. Oxford, U.K.: Springer, Jul. 2018, pp. 211–229.

[27] T. Sondag and H. Rajan, "A more precise abstract domain for multi-level caches for tighter WCET analysis," in *Proc. 31st IEEE Real-Time Syst. Symp.*, Nov. 2010, pp. 395–404.

[28] G. Stock, S. Hahn, and J. Reineke, "Cache persistence analysis: Finally exact," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2019, pp. 481–494.

[29] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *Proc. Int. Symp. Syst. Chip*, Dec. 2003, pp. 154–165.

[30] Q. Wan, H. Wu, and J. Xue, "Wcet-aware data selection and allocation for scratchpad memory," in *Proc. SIGPLAN/SIGBED Conf. Lang., Compil. Tools Embedded Syst.*, R. Wilhelm, H. Falk, and W. Yi, Eds. Beijing, China: ACM, Jun. 2012, pp. 41–50.

[31] T. Randall White, F. Mueller, A. Christopher Healy, B. David Whalley, and G. Marion Harmon, "Timing analysis for data and wrap-around fill caches," *Real-Time Syst.*, vol. 17, nos. 2–3, pp. 209–233, 1999.

[32] J. Whitham and N. Audsley, "Studying the applicability of the scratchpad memory management unit," in *Proc. 16th IEEE Real-Time Embedded Technol. Appl. Symp.*, Apr. 2010, pp. 205–214.

[33] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement. PLDI*, 1991, pp. 30–44.

[34] W. Zheng and H. Wu, "WCET-aware dynamic D-cache locking for a single task," in *Proc. 16th ACM SIGPLAN/SIGBED Conf. Lang., Compil. Tools Embedded Syst. CD-ROM - LCTES*, 2015, pp. 8:1–8:10.

JUAN SEGARRA received the degree in computer science and the Ph.D. degree from Universitat Jaume I, Spain, in 2003. In 2003, he joined the University of Zaragoza, where he is currently working with the Informática e Ingeniería de Sistemas Department. He is member of the Computer Architecture Group (gaZ), University of Zaragoza. His research interests include worst-case execution time and worst-case memory performance in hard real-time systems.

JORDI CORTADELLA (Fellow, IEEE) received the Ph.D. degree in CS from the Universitat Politècnica de Catalunya, Barcelona, Spain, in 1987. He is currently a Professor with the Computer Science Department, Universitat Politècnica de Catalunya. His current research interests include formal methods and computer-aided design of VLSI systems with a special emphasis on asynchronous circuits, concurrent systems, and logic synthesis. He is a member of the Academia Europaea. He received Best Paper Awards at ASYNC 2004 and 2016, DAC 2004, ACSD 2009, and FPGA 2020. He has served on the technical committees of several international conferences in the field of design automation and concurrent systems. He is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS.

RUBÉN GRAN TEJERO received the degree in computer science from the University of Zaragoza, Spain, and the Ph.D. degree from the Universitat Politècnica de Catalunya (UPC), in 2010. Since 2010, he has been working with the Informática e Ingeniería de Sistemas Department, University of Zaragoza. He is a member of the Computer Architecture Group (gaZ), University of Zaragoza. His research interests include worst-case in hard real-time systems, microarchitecture, and optimizing compilers for GPGPU's.

VÍCTOR VIÑALS-YÚFERA (Member, IEEE) received the M.S. degree in telecommunication and the Ph.D. degree in computer science from the Universitat Politècnica de Catalunya (UPC), in 1982 and 1987, respectively. He was an Associate Professor with the Facultat d'Informàtica de Barcelona, UPC, from 1983 to 1988. He is currently a Full Professor with the Informática e Ingeniería de Sistemas Department, University of Zaragoza, Zaragoza, Spain. His research interests include processor microarchitecture, memory hierarchy, and parallel computer architecture. He also belongs to the Juslibol Midday Runners Team and the Computer Architecture Group, University of Zaragoza. He is a member of ACM and the IEEE Computer Society.

• • •