IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 1 December 2022, accepted 14 December 2022, date of publication 16 December 2022, date of current version 27 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3230068

METHODS

Improving the Configuration of the Predictable ACDC Data Cache for Real-Time Systems

JUAN SEGARRA[1] **AND ANTONIO MARTÍ-CAMPOY**[2]
[1]Departamento de Informática e Ingeniería de Sistemas (DIIS), Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, 50009 Zaragoza, Spain
[2]Institute for Information and Communication Technologies (ITACA), Universitat Politècnica de València, 46022 Valencia, Spain

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

This work was supported in part by the Ministerio de Ciencia e Innovación (MCIN)/Agencia Estatal de Investigación (AIE)/10.13039/501100011033 under Grant PID2019-105660RB-C21 and Grant PID2020-117713RB-I00; and in part by the Government of Aragon, Research Group, under Grant T5820R.

ABSTRACT In real-time systems, analyzing the worst-case execution time (WCET) of a task in the presence of data caches is hard. The ACDC is a data cache that provides predictability, facilitating WCET analysis. It works by granting data cache replacement permission to specific load/store instructions. Nonetheless, knowing how to select these instructions to minimize the WCET, i.e., configuring the ACDC, is not trivial. In this paper, we propose four new methods to configure the ACDC, and compare them with existing methods. Unlike those in previous studies, our proposed methods provide specific ACDC configurations for the different phases of a given task, instead of a single ACDC configuration per task. We evaluate the WCET bounds obtained when using different ACDC configuration methods on the TACLeBench benchmark suite. Our results show that the most complex benchmarks work better with multiple-content configurations, which indicates that realistic tasks may also benefit from this kind of configuration. The methods proposed in this study improve the WCET in more than 60% of cases, with an average WCET improvement of nearly 5% and up to 50% in some cases.

INDEX TERMS ACDC, WCET, data cache, real-time, genetic algorithms, static analysis.

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, which perform the required functionality. These tasks can be organized by priorities and they have to be scheduled such that they meet their deadlines. To ensure the correctness of the system, tasks must be schedulable considering that each one requires its corresponding Worst-Case Execution Time (WCET) to be completed.

One of the main challenges in WCET analysis is the memory hierarchy [1]. General purpose systems usually have set-associative LRU instruction and data caches for the first level, and unified set-associative LRU caches for other levels. LRU caches introduce considerable uncertainty into WCET estimation, and having several levels of cache enlarges such uncertainty, and hence, they are not adequate for hard real-time systems. This leaves system designers with three options: disable the caches, losing the performance improvements brought by their use; carry out an analysis or measurement using conventional caches [2], usually complex and pessimistic; or instead use specialized caches that focus on predictability rather than performance [3]. Cache locking is the traditional way to increase cache predictability and simplify WCET estimation. Lockable caches can be used both for instruction [4] and data caches [5]. For data caches, there are other predictable hardware proposals, such as the Address-Cache Data-Cache (ACDC) [6]. The ACDC is a very small instruction-driven data cache able to exploit data reuse in a very controlled and predictable way, with a much better performance than a locked data cache. Like lockable caches, the ACDC must be configured for the task to be run, i.e., to specify how the cache must behave upon specific data requests. However, obtaining a configuration that minimizes

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

the WCET is not trivial. Existing approaches generate, for each task, a particular single-content configuration to be used during the execution of the whole task. The goal of this configuration is to reduce the WCET of the task, but no studies have yet evaluated the extent of such a reduction.

In this study, we focus on the ACDC. We compare the WCET resulting from different existing ACDC configuration approaches, and propose new configuration methods. Specifically, we propose two heuristic multiple-content configuration methods, that is, methods to obtain specific configurations for the different phases of a task. Apart from more accurate configurations, this avoids the restrictions that the limited size of the ACDC may imply. Additionally, we propose two genetic algorithms to further explore the configuration space, considering single and multiple content types, to test whether the configurations obtained can be improved. Evaluating how each of these proposals performs is a first step towards integrating any of them in a compiler.

Our contributions can be summarized as follows:

- Proposal of two heuristic multiple-content configuration methods to overcome the ACDC size limitation associated with existing (single-content heuristic) methods.
- Proposal of two genetic algorithms (single and multiple content) to evaluate how much room there is for improvement in both the existing single-content and our proposed multiple-content heuristic methods.
- Performance comparison and analysis of existing approaches and our proposed ones (both heuristic and genetic) in terms of WCET and analysis time.

With these contributions, given methods can be selected for specific tasks.

The rest of this paper is organized as follows. Section II describes related work, in particular, the ACDC data cache and existing configuration methods. Then, we propose new single-content and multiple-content methods in Section III and Section IV, respectively. Subsequently, Section V describes the experimental environment, and Section VI details our experiments and their results. Finally, we present our conclusions in Section VII.

II. RELATED WORK

Typical memory hierarchies found in general purpose processors are not adequate for real-time systems. For instance, a WCET analysis of a level 2 cache shared between different cores should consider that this hardware resource is always busy attending other cores, and therefore, that such a potential delay in the worst case may be worse than the delay in simpler architectures [7]. To reduce the complexity of WCET analyses of systems with cache memories, many studies propose using scratchpad memories or lockable caches[1] [3]. These caches are preconfigured/preloaded with specific instructions/data, and locked so that their content is not evicted. In this way, the hit/miss computation in the worst case is much easier and more precise, since it is not affected by the uncertainty introduced by the path taken during the execution of tasks.

There are two ways of using lockable caches: statically and dynamically. *Static locking* methods lock select cache content belonging to *all the tasks* that run in the system, this content being fixed at system start-up [8], [9]. On the other hand, *dynamic locking* methods select content to lock once or more per task. In general, dynamic locking performs better than static locking in terms of WCET [10]. Focusing on dynamic locking, *single-content dynamic locking* selects a single set of content per task, which is loaded and locked at each task context switch (e.g. [11]), and *multiple-content dynamic locking* refers to methods that allow each task to load and lock different cache contents at will during its execution (e.g. [4], [12]).

Cache locking methods work especially well for instruction caches [13]. In contrast, their application to data caches has major drawbacks that appear 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. Therefore, the application of cache locking to data caches is restricted to load and lock in cache whole data structures for specific chunks of code [5].

Seeking to overcome the aforementioned limitations, other data cache designs have been proposed. These include the ACDC, which is a predictable data cache where only certain instructions (previously configured) are allowed to replace cache lines. The Fully-Associative FIFO tagged Buffers (FAFBs) are auxiliary caches that follow a similar approach, but using cache lines organized as FIFO buffers [14]. FAFBs are designed to work coupled with any level 1 data cache and focus exclusively on exploiting data accesses to arrays with temporal reuse, commonly generated by tiling transformations in the compiler. In particular, FAFBs manage these accesses, avoiding the associated pollution of the main cache, and let the main cache deal with any other data access.

In this paper, we focus on the ACDC, described below, since it exploits most reuse types and does not need any particular compiler optimization.

A. ACDC DATA CACHE OVERVIEW

The ACDC is a small instruction-driven data cache that effectively exploits reuse [6]. It operates from a fixed preselected subset of load/store instruction addresses held in the AC part of the ACDC cache (see Figure 1). These selected load/store instructions have data cache replacement permissions (DRPs). Each permission is associated with a particular

[1]Lockable caches are present in processors of most manufacturers, including those of Motorola (ColdFire, PowerPC, MPC7451, MPC7400), MIPS (MIPS32), ARM (904, 946E-S), Integrated Device Technology (79R4650, 79RC64574), and Intel (960). Scratchpad memories are also common, being found, for instance, in the Xilinx Zynq Ultrascale+.

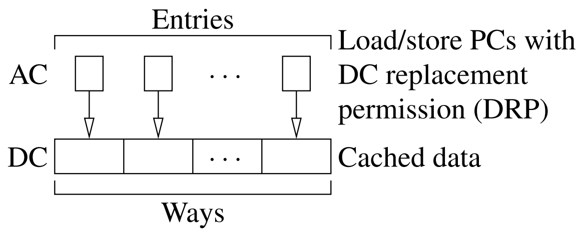

FIGURE 1. ACDC schema [16].

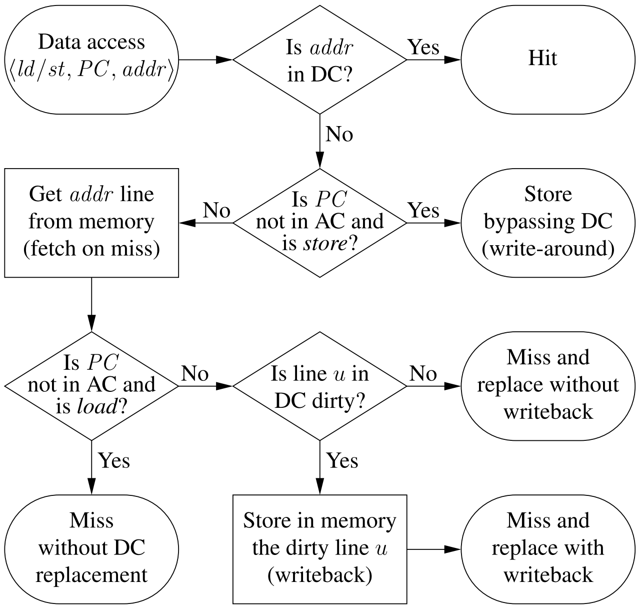
Data access ⟨ld/st, PC, addr⟩
Is addr in DC?
Yes
Hit
No
Get addr line from memory (fetch on miss)
No
Is PC not in AC and is store?
Yes
Store bypassing DC (write-around)
Is PC not in AC and is load?
No
Is line u in DC dirty?
No
Miss and replace without writeback
Yes
Yes
Miss without DC replacement
Store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

FIGURE 2. ACDC operation flow chart for a *load*/*store* instruction (in *PC*) to *addr*, which may evict a cache line *u*. *PC*s in AC are those with DC replacement permission [16]. Note that the first three decisions can be evaluated in parallel.

data cache line in the DC part of the ACDC. Thus, when executing a load/store instruction that misses in the DC, the replacement of the data line on the DC will be only allowed if the instruction has DRP (i.e., the PC of the 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. In other words, using the ACDC the programmer does not need to partition data sets to reduce cache conflicts. Figure 2 is a flowchart of the ACDC's behavior. For data accesses, look-up is fully-associative, meaning that any access may benefit from the cached content. On a miss, if the missing load/store has DRP (its PC is in the AC), the DC line assigned to this load/store is replaced, as in a conventional write-back write-miss-allocate cache. However, misses triggered by instructions without DRP bypass the DC. That is, loads bring the specified data to the processor without modifying the DC, and stores write directly to main memory without fetching the missing data, as in a write-around cache [15]. This behavior provides much better performance than a locked data cache [6].

As can be seen, only the set of instruction addresses with replacement permission in the ACDC can evict its content, and each one of these instructions can only replace its own associated cache line. Thus, hits and misses depend exclusively on whether the referenced data have been previously accessed by a memory instruction with replacement permission. A data reuse analysis of the task provides this information [17].

In order to support preemptive multitasking real-time systems, cache-related preemption delay (CRPD) must be considered. Calculating such a delay value is also straightforward with the ACDC. Given its small size (from 256 B to 2 KiB in this study), saving and restoring the whole cache content (AC and DC) is acceptable [6]. This procedure can be used independently of the configuration policy, i.e., a single-content configuration for the whole execution of a task, or specific configurations for the different phases of a task (multiple-content locking). Additionally, the resulting CRPD is constant, and it can be easily integrated into the response time computation [6].

Configuring the ACDC means selecting the set of load/store instructions that will have DRP (those to be held in the AC part of the ACDC). Previous studies have proposed two ACDC configuration methods: single-content WCET optimization for single-path tasks (S-OSP) [6] and single-content estimation of WCET benefit from potential DRPs (S-EB) [16]. Both of these methods obtain a single set of load/store instructions for a given task. These instructions are preloaded before running the task, and also when the task resumes its execution after a context switch.

B. SINGLE-CONTENT WCET OPTIMIZATION FOR SINGLE-PATH TASKS (S-OSP)

This method was proposed along with the ACDC itself [6]. It is an extension of the Lock for Maximizing Schedulability (Lock-MS) method, which minimizes the WCET in presence of lockable instruction caches [11]. This method builds a structure-based Integer Linear Programming (ILP) model of the task, instead of the more common ILP flow model used in the Implicit Path Enumeration Technique (IPET) [18]. Structure-based models are faster to solve, but they have more limitations than the IPET regarding costs that depend on previously taken paths. Therefore, these methods work well for locked instruction caches, but not for LRU caches. For systems with an ACDC, this method is only valid if the data memory access costs of the task analyzed do not depend on the path taken, i.e., when the task has a single path or a set of alternative non-interfering paths. Otherwise, the results may not be optimal, that is, the obtained selection of load/store instructions to be granted DRP may not be the one that provides the best WCET bound.

This method obtains the ACDC configuration by solving an ILP model, summarized as follows. The cost of each particular path p in the task is composed of the ACDC oper-

TABLE 1. Variables and constants for the integer linear programming models.

Constant	Description
$me_{pc(ref)}$	Maximum number of times that the instruction performing ref can be executed
hc	Data hit cost
mc	Data miss cost
wbc	Write-back cost
mm_{ref}	(in S-OSP) Maximum number of misses of an array reference ref in a loop ($mm_{ref} = ACDCmisses_{ref}$ in [6])
Variable	**Description**
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
drp_{PC}	1 (true) if PC is granted data replacement permission; 0 otherwise
mm_{ref}	(in S-EB) Maximum number of misses of an array reference ref in a loop ($mm_{ref} = k \cdot \sum_{d \in EE(loop(ref))} d$ in [16])

ation costs $acdcCost_p$ plus other costs in the path (see the details of the Lock-MS method for the costs not related to the ACDC [11]). In addition, the WCET must be larger than or equal to the cost of any of the paths, plus the cost of configuring the ACDC ($preloadCost$) [6].

$$\begin{aligned} rClWCET &= preloadCost + worstPath \\ worstPath &\geq acdcCost_p + otherCosts_p, \ \forall\, p \in Paths. \end{aligned} \tag{1}$$

Then, the minimization of $WCET$ is set as the objective of the model. That is, it will try to reduce the WCET as much as possible, by selecting the loads/stores to be granted DRP. Each load/store (located at a given address, PC) is associated with a binary variable stating whether this instruction is granted DRP or not (drp_{PC}). Hence, the sum of these variables provides the number of PCs to preload in the ACDC, which must be less than or equal to its number of entries (AC entries in Figure 1) [6].

$$\begin{aligned} rClloadedDRPs &= \sum_{PC} drp_{PC} \\ loadedDRPs &\leq acdcEntries \end{aligned} \tag{2}$$

Since this method calculates bounds on paths, the overall memory cost for a path p ($acdcCost_p$) can be calculated as the sum of the cost of each particular memory reference ref (associated with a given static load/store instruction) in the path. The access cost of a given reference ref is the sum of its number of occurrences (number of data hits dh_{ref}, misses dm_{ref}, and write-backs wb_{ref}) times the cost of each occurrence, respectively (see Table 1) [6].

$$rClacdcCost_p = \sum_{ref \in memRefs_p} hc \cdot dh_{ref} + mc \cdot dm_{ref} + wbc \cdot wb_{ref} \tag{3}$$

In turn, the number of occurrences can be calculated depending on the DRPs (binary drp_{PC} variables) and certain data reuse information [6].

$$\begin{aligned} rCldh_{ref} &= me_{pc(ref)} - dm_{ref} \\ wb_{ref} &= \begin{cases} dm_{ref} \cdot drp_{pc(ref)} & \text{if } ref \text{ has group-reuse stores,} \\ 0 & \text{otherwise.} \end{cases} \\ dm_{ref} &= mm_{ref} \cdot drp_{pc(ref)} \\ &\quad + me_{pc(ref)} \cdot (1 - drp_{pc(ref)}) \end{aligned} \tag{4}$$

Note that the only actual variables are drp related. The other values are either constants obtained at an earlier stage of analysis, or calculations based on the drp variables.

As stated above, the limitation of this method is the presence of interfering paths, such as those found in *if-then-else* statements inside loops. This method assumes that each branch of a conditional is always executed in the same way, either always taken, or always not-taken.

C. SINGLE-CONTENT ESTIMATION OF WCET BENEFIT FROM POTENTIAL DRPs (S-EB)

S-EB is a more recent proposal that works by estimating the specific WCET benefit from granting DRP to each and every load/store instruction [16]. As in S-OSP, intereferring paths are problematic. However, instead of isolating paths like S-OSP, S-EB isolates each estimated benefit, i.e., it assumes that granting replacement permission to a given memory instruction does not modify the benefit estimated for other instructions. Once all benefits have been estimated, the instructions which provide the greatest benefit in terms of WCET reduction are selected. Since the WCET also depends on the relations between the selected DRPs and the paths in the task, this method does not guarantee an optimal configuration. Nevertheless, an ACDC with this configuration method provides better WCETs than typical (much larger) LRU data caches [16]. As far as we know, no studies have compared S-EB with S-OSP.

A detailed description of the ILP model for the S-EB method has been published previously [16]. First, an IPET model must be built to calculate the WCET by considering always-miss on data [18]. Then, further calculations are included in the model to estimate the potential benefit (b_{PC}) of granting DRP to each load/store instruction, as described below. These additional calculations do not affect the existing constraints. Essentially, for each instruction eligible to be granted DRP, the potential benefit is calculated as the sum of the cost of preloading this instruction in the AC ($preload$), the cost of accessing memory if granted DRP ($access$), the benefits for other instructions reusing the data cached by this load/store ($reuse$), and the cost of potential write-backs ($writeback$) [16].

$$b_{PC} = preload + \sum_{ref \ in \ PC} access_{ref} + reuse_{ref} + writeback_{ref} \tag{5}$$

The $preload$ cost is constant, depending on the hardware. The memory access benefit ($access_{ref}$) of an instruction with

DRP depends on whether it is an access to a scalar outside a loop (no benefit because it will miss anyway), or an access inside a loop. Accesses to a scalar in a loop will always hit except the first time, and therefore, all miss costs in the always-miss model must be replaced by a hit cost $(hc - mc)$ minus one. Hits/misses of accesses to an array in a loop depend on self-spatial reuse, which must be previously calculated to generate mm_{ref} (see Table 1) [16].

$$access_{ref} = \begin{cases} 0 \quad \text{if } ref \text{ is (scalar) outside loop,} \\ (hc - mc) \cdot dm_{ref} + (mc - hc) \\ \quad \text{if } ref \text{ is scalar inside loop,} \\ (hc - mc) \cdot dm_{ref} + (mc - hc) \cdot mm_{ref} \\ \quad \text{if } ref \text{ is array inside loop.} \end{cases} \tag{6}$$

For memory accesses reusing content brought from a previous instruction, assuming DRP for this previous instruction implies considering a hit cost instead of a miss cost $(hc - mc)$ for all the times that this content is reused (dm_r) [16].

$$reuse_{ref} = (hc - mc) \cdot \sum_{r \text{ reusing } ref} dm_r \tag{7}$$

Finally, the number of write-backs depends on whether the cached data has been modified (i.e., has stores with group reuse) or not [16].

$$writeback_{ref} = \begin{cases} wbc \cdot mm_{ref} & \text{if } ref \text{ has group reuse stores,} \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

III. SINGLE-CONTENT GENETIC ALGORITHM (S-GA)

As detailed in the related work section, two methods may be used to configure the ACDC: S-OSP (Section II-B) and S-EB (Section II-C). The WCET results obtained using an ACDC configured with S-EB are better than those obtained in the presence of a conventional LRU data cache [16]. However, no studies have evaluated the performance of these configuration methods, or compared them to determine which performs better. In this section, we propose a new method, based on genetic algorithms, for exploring the solution space of ACDC configurations, to enable us to assess whether S-OSP and S-EB are good enough, or there is room for improvement.

Genetic algorithms are well known metaheuristics used to solve optimization problems [19]. Although there is no guarantee that these algorithms find the optimal solution, they usually provide at least a suboptimal solution in a practical amount of time. They rely on the iteration of transformations applied to an initial set of possible solutions. This set of solutions is the *population*, and each solution is an *individual*. The transformations resemble biological operators that allow a population to evolve, combining and changing the individuals. In this study, the purpose of S-GA is to improve on the WCET obtained by S-EB, the output of which will be included in the initial solution. S-GA is composed of the following elements and operators:

1) Input data: list of PCs, corresponding to load/store instructions in the task; number of ACDC entries, corresponding to the number of PCs that can be kept in the ACDC; and list of PCs to be kept in the ACDC (those with DRP), selected by the heuristic algorithm S-EB. The number of PCs in this second list must be less than or equal to the number of ACDC entries.
2) Representation: each individual is an array of *acdcEntries* elements, that is, all individuals have an equal and fixed number of elements, corresponding to the number of entries in the AC part of the ACDC. Each element stores the address (PC) of a load/store instruction. The presence of a PC in this array means that this PC will be loaded in the AC and then allow the replacement of its corresponding content in the DC part of the ACDC. Possible solutions may use fewer AC entries than available because loading PCs in the AC has a cost in the WCET (*preloadCost*) that may not be compensated for later during the task execution. Since the size of the array is set to the number of AC entries for all individuals in order to keep the crossover operator as simple as possible, we use replicated elements in the array (PC values) to indicate invalid/empty AC entries. That is, to deal with unused AC entries, PCs in the individual are replicated as many times as required until the individual is full. Once the final solution is obtained, replicated PCs are removed from the solution.
3) Initialization: The PCs selected by the S-EB method are the most valuable for reducing the WCET, but they may represent a local optimum. To expand the search space, the initial population is created in three different ways:
    a) The first individual is filled with the PCs provided by S-EB. If the number of PCs is smaller than the number of AC entries *acdcEntries*, the PCs are replicated until reaching the size of the AC.
    b) The second individual is filled with *acdcEntries* PCs chosen randomly from the list of PCs in the task. In this second individual, replicated PCs may appear.
    c) Loading PCs in the AC has a cost that may increase the WCET instead of reducing it. To check whether partially loading the AC improves the WCET, for the rest of the individuals, n PCs are randomly chosen from the list of PCs of the task, where n is a random value between 1 and the number of AC entries. The rest of the array of each individual is filled by replicating some of these PCs.
4) Fitness function: the fitness of each individual is the WCET bound estimated by solving the ILP model of the task using the individual (without repeated PCs) as the ACDC configuration.
5) Selection and crossover: selection is accomplished using a deterministic binary tournament, and elitism is applied so that the best individual is always incorporated to the new generation. One point crossover is carried out splitting the parents at a randomly chosen point, and then merging them to create two new

off-spring. A crossover probability is applied to decide whether the parents are merged or pass directly to the new generation.

6) Mutation: when a PC of an individual is randomly selected to mutate, it can mutate in three different ways with the same probability. In the first type of mutation, the PC can be removed from the individual, that is, the number of PCs is reduced. Removing a PC is carried out by overwriting the PC with a copy of any other PC value in the array to mark it as invalid. This mutation might have no effect. In the second type of mutation, the PC is replaced with a PC coming from the list of PCs in the task, and therefore, the number of DRPs remains unchanged. In the last type of mutation, a new PC is added to the individual, if possible, increasing the number of DRPs. When adding a new PC to the individual, either replacing an existing PC or just adding one more DRP, the presence of the new PC in the individual is not checked, and therefore, this mutation may have no effect.

The S-GA parameters, like population size, finishing condition, and crossover and mutation probabilities are described in Section V-B.

IV. MULTIPLE-CONTENT ACDC CONFIGURATION

The main drawback of single-content dynamic locking methods in lockable instruction caches is their limited ability to adapt to changes in the working data set that may appear during the different phases in the execution of a program. The ACDC is far more flexible, due to the fact that it does not lock data contents, but rather sets replacement permissions. Nonetheless, programs that work with multiple data structures on different parts of the code still suffer from the aforementioned limitations.

An apparently straightforward improvement is to define region-specific contents to load and lock prior to the execution of these regions in the task. Such an approach has been used for lockable instruction caches [4], [12], [20], [21]. However, it implies finding both adequate loading points and adequate elements to lock at each point, taking into account the time overhead of instruction loading and locking, and the modification of the memory layout of the task. Thus, improving the behavior of single-content dynamic locking is not as straightforward as one might think.

In this section, we detail several proposals to extend previous single-content ACDC configuration methods (S-OSP, S-EB, and S-GA) to multiple content. As in previous studies on lockable instruction caches [4], we place the ACDC reconfiguration points (i.e., points to revoke DRPs or grant new ones) at the beginning of the program and prior to the outer loops.

Definition 1: Let us define *outerLoops* as the set of loops not nested, i.e., not included in any other loop. Loops in functions called from the body of a loop are considered nested loops. Recursive loops (loops in functions that call themselves from within their own loop) can be processed by, for instance, inlining the first function call.

Reconfiguring DRPs before entering these loops addresses the following points. First, since spatial reuse (e.g., an array traversal) is found in loops, each one of these loops has a specialized configuration and a larger share of the ACDC. Second, the DRPs loaded at the beginning of the program address the temporal reuse cases that may cover the whole program (global variables and part of the stack). Third, all reconfigurations occur outside loops, and therefore, their corresponding overhead will never be multiplied by the number of loop iterations. As in previous studies, we assume that reloading just a part of the ACDC is possible, that is, if only one subset of the DRPs of the ACDC needs updating, it can be updated without affecting the other DRPs. The behavior of context switches does not differ with respect to single-content configurations, that is, the ACDC (both the AC and DC parts) may be saved at context switches and restored when the task resumes its execution.

A. MULTIPLE-CONTENT WCET OPTIMIZATION FOR SINGLE-PATH TASKS (M-OSP)

Our first multiple-content ACDC configuration method is built on top of the single-content WCET optimization for single-path tasks outlined in Section II-B [6]. S-OSP defines the DRP of each load/store as a binary variable indicating whether such permission is granted or not. Depending on the constraints and the constant cost of each possible event, the solver calculates the specific values of the DRP variables that minimize the function modeling the WCET bound of the task.

In order to manage multiple ACDC configurations, M-OSP adds the following constraints to the S-OSP model. First, the total number of entries of the ACDC is divided into two sets.

$$rCl\, staticDRPs + dynamicDRPs \leq acdcEntries \tag{9}$$

The DRPs in the static set will be loaded before the task begins its execution and will not be revoked during the execution of the task. Essentially, these DRPs should contain the addresses of the first loads/stores accessing global scalar variables or accessing certain positions of the stack (parameters or scalar variables) for the first time. Further, instructions granted these DRPs cannot ever be inside loops (see Definition 1).

$$\begin{aligned} &rCl\, globalDRPs = staticDRPs \\ &\quad globalDRPs = \sum_{PC \in \{pc(ref)\}} drp_{PC}, \ \forall\, ref \text{ outside } outerLoops \end{aligned} \tag{10}$$

The DRPs in the second set (*dynamicDRPs*) will not be loaded before the task begins its execution, but will be updated before each outer loop. Hence, the number of DRPs in any outer loop l must be less than or equal to the entries devoted to this set of DRPs in the ACDC.

$$rCl\, loopDRPs_l \leq dynamicDRPs, \ \forall\, l \in outerLoops \tag{11}$$

Essentially, DRPs of each outer loop l should contain addresses of loads/stores inside the corresponding loop l or its nested loops.

$$rCl\, loopDRPs_l = \sum_{PC \in \{pc(ref)\}} drp_{PC}, \ \forall\, ref \text{ inside } l \quad (12)$$

Finally, the loading cost of the ACDC can be calculated as the constant cost related to the function calls *loadingCalls* to load the DRPs plus the variable cost *actualLoads* depending on how many DRPs have been loaded. The cost of the function calls is the number of outer loops times the cost per call. The cost of the actual loads is the main memory latency times the number of loaded DRPs.

$$\begin{aligned} rCl\, preloadCost &= loadingCalls + actualLoads \\ loadingCalls &= |outerLoops| \cdot callCost \\ actualLoads &= memLatency \cdot loadedDRPs \\ loadedDRPs &= \sum_{PC} drp_{PC} \end{aligned} \quad (13)$$

Note that, for simplicity, we assume that loading the configuration for the first outer loop is performed in the same way as for any other loop. Nonetheless, this first loop configuration could be set along with the global DRPs and thus avoid its loading call. We also assume, for simplicity, that all loading calls have the same cost, though a specific constant cost could be used for each loading function call.

Once these constraints have been set, the ILP model is solved, to obtain the values for the drp_{PC} variables (1 or 0 indicating whether the corresponding DRP is granted or not) that optimize the model. Using the selected DRPs, the WCET can be analyzed [16]. Recall that the selected DRPs may not be the ones that minimize the WCET, since the optimized model assumes that the data accesses do not depend on the paths taken in the task. That is, the selected DRPs are only the best possible ones in the case of single-path tasks.

B. MULTIPLE-CONTENT ESTIMATION OF WCET BENEFIT FROM POTENTIAL DRPs (M-EB)

This method requires a priori estimation of the benefit of granting DRP to each suitable load/store instruction, like S-EB (Section II-C). However, instead of just selecting the n ($\leq acdcEntries$) best instructions to which to grant DRPs for the whole program execution, a new ILP model is built to select the DRPs. This new model is based on the same concepts and has the same constraints as M-OSP.

$$\begin{aligned} rCl\, acdcEntries &\geq staticDRPs + dynamicDRPs \\ globalDRPs &= staticDRPs \\ globalDRPs &= \sum_{PC \in \{pc(ref)\}} drp_{PC}, \ \forall\, ref \text{ outside } outerLoops \\ loopDRPs_l &\leq dynamicDRPs, \ \forall\, l \in outerLoops \\ loopDRPs_l &= \sum_{PC \in \{pc(ref)\}} drp_{PC}, \ \forall\, ref \text{ inside } l \end{aligned} \quad (14)$$

On the other hand, unlike with the S-OSP, these constraints are not integrated into a model that includes the task. In this case, the ILP model just contains the aforementioned constraints, plus the objective function. This function must maximize the benefit of the selected DRPs, and hence, it can be set as the sum of the constant estimation of the benefit of granting DRP to the load/store at PC (b_{PC}) times the binary variable drp_{PC}.

$$rCl \max: \sum_{PC} b_{PC} \cdot drp_{PC} \quad (15)$$

Thus, solving this ILP model provides the values for the drp_{PC} variables that maximize the benefit, i.e., the PCs to load into AC. As in the S-EB method, M-EB provides the best ACDC configuration based on isolated estimates, and therefore, the configuration it provides is not necessarily the one that minimizes the WCET.

C. MULTIPLE-CONTENT GENETIC ALGORITHM (M-GA)

For the multiple-content version, the single-content S-GA has been modified in the way the solution is represented. An individual is now a $(1 + |outerLoops|) \times acdcEntries$ matrix, where each row represents an ACDC configuration point in the task, and the elements of the row are the PCs that are granted DRPs at the corresponding configuration point. The first row represents the initial configuration for the task, with PCs with DRPs for the whole task execution. The other rows represent the DRPs in the different *outerLoops* in the task. The number of distinct PCs in the first row, *staticDRPs* ($< acdcEntries$), is fixed for each experiment. For the other rows, the number of distinct PCs is $acdcEntries - staticDRPs$. Other genetic operators, like selection, crossover and mutation, perform in the same way as in S-GA, but considering only the distinct PCs in each row.

V. EXPERIMENTAL ENVIRONMENT

We use an ARMv7 architecture with a typical 5-stage in-order pipeline (instruction fetch, decode, execute, data memory access, and write-back). Since we focus on the memory hierarchy, we assume that the pipeline never stalls, and all its stages are completed in a single cycle, except those involving the instruction/data cache or memory. We use an ACDC (4 to 32 entries, 256 B to 2 KiB) as a data cache and a locked 8-way set-associative instruction cache (64 sets, 32 KiB). Using a locked instruction cache preloaded with specific contents for each benchmark enables realistic results and avoids the uncertainty that an LRU instruction cache would introduce. Cache hits take a single cycle, and misses take one look-up cycle plus the cycles required for a memory access. We consider a memory access time of 13 cycles both for instructions and data, which is a realistic value for main memories such as the Automotive DRAM MT46V16M16 [22] clocked at 100 MHz, and has been used in previous studies [16], [23]. For the ACDC, accesses that replace a dirty cache line take a total of 27 cycles to look-up (1), write back the dirty line to memory (13), and bring in the new line from memory (13).

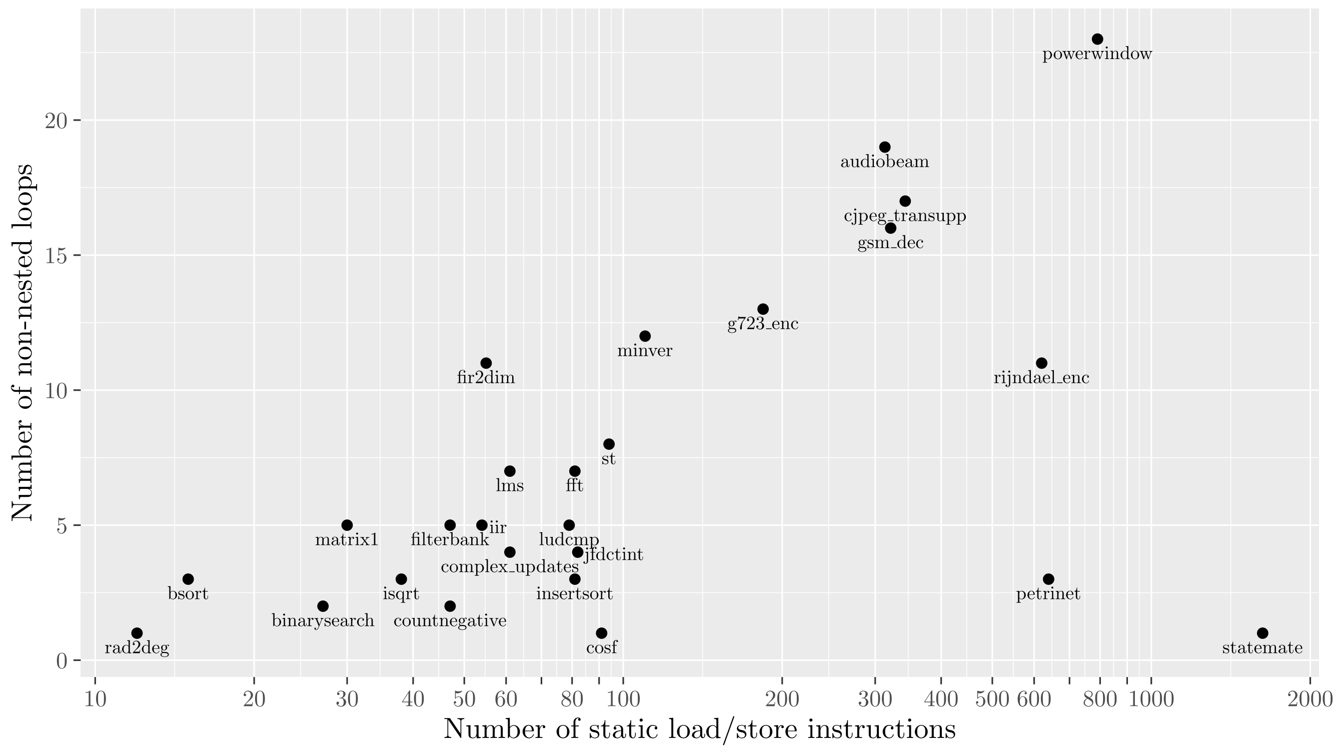

FIGURE 3. Tested benchmarks.

For an instruction with multiple accesses (push/pop), requests are processed sequentially in an isolated way, i.e., no burst memory transfers are considered.

A. BENCHMARKS

We use a subset of benchmarks from TACLeBench [24]. They have been compiled with gcc 9.2.1, with level 2 optimization (-O2), since this optimization level provides the best WCET results in general [16]. The binary files are analyzed using *angr* version 9.0.4663 [25]. Recursive benchmarks are not yet supported by the analysis tools considered, and have therefore been excluded. In addition, benchmarks whose control flow graph (CFG) is not correctly generated by *angr* have been excluded. Further, we have excluded benchmarks *deg2rad* and *rijndael_dec* because their results are almost identical to those of *rad2deg* and *rijndael_enc*, and benchmark *cover* because it has very few memory operations, and none of them inside a loop. For each benchmark, the maximum number of iterations in loops has been manually set according to the annotations in source files and carefully observing the transformations performed by the compiler. Nevertheless, automatic methods could be used [26].

To provide an overview of the benchmarks used in our study, they are plotted on Figure 3. The horizontal axis shows the number of static load/store instructions in the benchmark, as a *data complexity* metric, this being used to order the benchmarks in the following figures in this paper. The vertical axis shows the number of outer loops (see Definition 1), recalling that the ACDC reconfiguration points for multiple-content configurations are placed before these loops. Overall, Figure 3 provides some insight into the solution space to explore, both for single-content (x-axis) and multiple-content (the whole figure) configurations. For instance, note that *rad2deg*, *cosf*, and *statemate* have a single non-nested loop, and hence, according to our criteria for placing ACDC reconfiguration points (Section IV), these benchmarks are not considered for multiple-content ACDC configurations. In general, the benchmarks considered cover wide ranges of load/store instructions and numbers of outer loops.

B. GENETIC ALGORITHM SETUP

For S-GA execution, we use the parameters in Table 2, tuned after an exploratory set of runs for a subset of the benchmarks. Hereafter, we call each ⟨benchmark, ACDC entries⟩ pair an *experiment*. A total of 104 experiments (26 benchmarks times 4 ACDC entries) are performed. Since the genetic algorithm uses the pseudo-random number generator arc4random from stdlib in several operators, the effect of the seed may be significant. To identify this effect, the algorithm runs 10 times for each experiment using the four ACDC sizes, yielding a total of 1040 executions (26 benchmarks $\times$ 4 cache sizes $\times$ 10 repetitions).

For M-GA, the number of experiments to check all possible ACDC configurations is much larger than for S-GA. Therefore, a first exploration has been carried out with a

TABLE 2. Parameters of S-GA and M-GA.

Population:	99
Finish condition:	200 generations run
Crossover probability:	0.6
Mutation probability:	0.001
ACDC entries:	4, 8, 16 and 32

smaller population and fewer generations. In this exploration, we have observed that the distribution of ACDC entries into *staticDRPs* + *dynamicDRPs* in the best result always coincides with the one heuristically chosen by M-EB. Therefore, to provide M-GA results comparable to those obtained by S-GA, we use the same parameters in both genetic algorithm methods (Table 2), and restrict the solution space of M-GA to the distribution of ACDC entries provided by M-EB.

VI. RESULTS

In this section, we first analyze the behavior of the genetic algorithms, especially the effect of the pseudo-random number generator. In addition, we study the execution time of the genetic algorithms and its relationship with ACDC size and benchmarks. Afterwards, we analyze the WCET bounds provided by each method.

A. S-GA RESULTS

Table 3 shows several WCET bounds obtained by the S-GA ACDC configuration method. It shows the average, minimum and maximum values of the ten repetitions for each experiment, ⟨benchmark, ACDC entries⟩. The main statistics for $\min(WCET)/\max(WCET)$ are listed in Table 4, and Figure 4 shows the standard deviation within the ten repetitions for each of the 104 experiments. Both Table 4 and Figure 4 show that the performance of the S-GA is very similar in most of the experiments, with no effect from the pseudo-random number generator. That is, the resulting WCET is the same or extremely similar for the ten repetitions. Nonetheless, in some cases, the effect of the pseudo-random number generator is larger and, for the same experiment, the S-GA provides different values of WCET. This is the case of *cosf* for ACDC sizes of 8, 16 and 32, *filterbank* for ACDC sizes of 8 and 16, and *fft* for an ACDC size of 8. These benchmarks perform rather complex scientific calculations, using codes that offer a really large number possibilities for configuring the ACDC. In particular, the *cosf* benchmark contains many conditional statements that change the flow control, and hence, S-EB generates suboptimal configurations; while the *filterbank* benchmark contains many loops with array traversals, which also implies many candidates for the granting of replacement permissions; and the *fft* benchmark contains both conditional statements and arrays in loops. Thus, for these benchmarks, S-GA begins with an initial solution (the configuration generated by S-EB) that has considerable room for improvement, which makes it much more dependent on the random generation of new individuals.

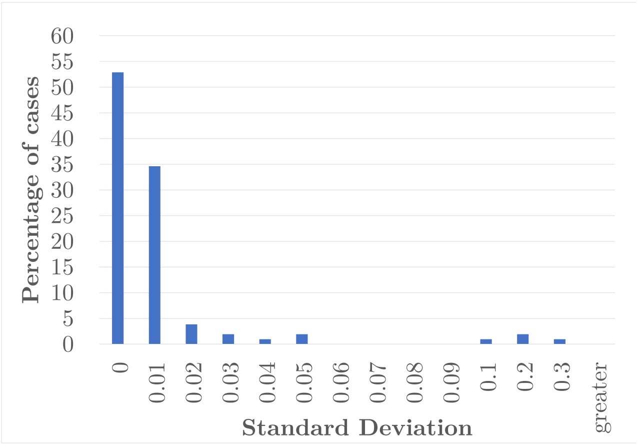

FIGURE 4. Histogram of the standard deviation (WCET obtained by S-GA) within the ten runs of each experiment ⟨benchmark, ACDC entries⟩.

B. M-GA RESULTS

Table 5 shows the average, minimum and maximum values of the ten repetitions for each ⟨benchmark, ACDC entries⟩ experiment. As above, each experiment consists of 200 generations of 99 WCET analysis, each one using the ACDC configuration provided by M-GA in this case. Table 6 shows the main statistics for $\min(WCET)/\max(WCET)$, and Figure 5 shows the standard deviation within the ten repetitions for each of the 92 experiments. Table 6 and Figure 5 show that in most cases the performance of M-GA is the same or very similar, with no effect from the pseudo-random number generator, although there is a slightly larger dispersion than in S-GA. The benchmarks showing the greatest variability within repetitions for some ACDC entries are *filterbank* for ACDC sizes 8, 16 and 32, and *ludcmp* for ACDC size 8. Like *filterbank*, the *ludcmp* benchmark has many array traversals in loops. However, it has few data structures and a large ACDC is not required, and hence, the variability is focused on one particular ACDC size.

The sensitivity of the genetic algorithm with respect to the pseudo-random number generator is low for most of the benchmarks, and similar for S-GA and M-GA. Nevertheless, the WCET analysis below uses the average WCET bound of the ten repetitions, to allow a fair comparison with other ACDC configuration methods.

C. EXECUTION TIME OF M-GA

Generating an ACDC configuration is very fast. It takes no more than a second, even though the proposed heuristic approaches work by building an ILP problem and then solving it. On the other hand, genetic algorithms require much more time, since they do not calculate a single ACDC configuration but rather multiple configurations, 19800 in our experiments (population times generations run). Moreover, for each ACDC configuration, both S-GA and M-GA perform a WCET analysis to guide the selection, which takes most of the required execution time. Nonetheless, in this case, both

TABLE 3. WCET bounds (average, minimum and maximum of ten repetitions, in cycles) obtained using S-GA as the ACDC configuration method.

	4 entries			8 entries			16 entries			32 entries		
Name	Average	Min.	Max.	Average	Min.	Max.	Average	Min.	Max.	Average	Min.	Max.
audiobeam	103326	103326	103326	97129	96995	97576	92400	92193	92540	89346	88944	89827
binarysearch	1283	1283	1283	1257	1257	1257	1257	1257	1257	1257	1257	1257
bsort	7322	7322	7322	7322	7322	7322	7329	7322	7335	7331	7322	7335
cjpeg_transupp	43890	43890	43890	25183	25183	25183	18085	18085	18085	15384	15355	15407
complex_updates	5366	5366	5366	4443	4443	4443	3039	3039	3039	1895	1895	1895
cosf	24478	23751	26612	18843	17561	22712	16818	10131	17561	11624	10131	17561
countnegative	1769	1769	1769	1743	1743	1743	1743	1743	1743	1743	1743	1743
fft	270345	270345	270345	183681	164343	190213	97383	97224	97536	97228	97224	97237
filterbank	13565	13565	13565	8954	8521	11407	6392	6064	6870	5917	5869	6298
fir2dim	7310	7310	7310	4476	4476	4476	3175	3163	3202	3047	3046	3059
g723_enc	117468	117468	117468	98344	96551	99866	79494	79339	79716	77143	76888	77222
gsm_dec	51989	51855	52401	47326	46915	47643	42681	42235	43152	38519	38269	38650
iir	2555	2555	2555	1853	1853	1853	1697	1697	1697	1697	1697	1697
insertsort	2099	2091	2130	1903	1883	1974	1861	1857	1870	1870	1857	1896
isqrt	4118	4118	4118	4092	4092	4092	4100	4092	4105	4144	4092	4183
jfdctint	5062	5062	5062	4256	4256	4256	3650	3632	3684	3151	3151	3151
lms	20766	20766	20766	19856	19856	19856	19729	19726	19739	19727	19726	19739
ludcmp	2395	2394	2407	2276	2238	2329	2200	2186	2251	2206	2186	2251
matrix1	2850	2850	2850	1576	1576	1576	1579	1576	1589	1602	1576	1719
minver	3728	3728	3728	3559	3559	3559	3368	3325	3429	3267	3247	3299
petrinet	17279	17279	17279	17019	17019	17019	16511	16499	16538	15492	15459	15524
powerwindow	11328185	11328185	11328185	10400099	10400099	10400099	9701591	9669551	9710572	8935163	8897103	8967374
rad2deg	210	210	210	210	210	210	210	210	210	210	210	210
rijndael_enc	1071960	1071960	1071960	1064056	1064056	1064056	1056711	1056711	1056711	1047527	1047247	1047676
st	106974	106974	106974	55104	55104	55104	54362	54362	54362	54362	54362	54362
statemate	761361	761361	761361	721139	721139	721139	664865	664208	664970	598339	595654	600273

TABLE 4. Statistics for min(*WCET*)/ max(*WCET*) for S-GA.

Items:	104
Average:	0.98
Minimum:	0.58
Maximum:	1.00
Standard Deviation:	0.07

S-GA and M-GA are executed to design the system, and therefore, long execution times are acceptable, as the better the design, the better the performance of the resulting system. Since both S-GA and M-GA have very similar execution times, only the ones for M-GA are discussed.

For our experiments, we use a 2.20 GHz Intel Xeon Gold 5120 CPU with 55 cores, with one experiment per core. This means that each experiment is run sequentially using a single core. Figure 6 shows the scatter plot of the analysis time of M-GA for the ten repetitions of each benchmark and the four sizes. As can be seen, the 40 points per benchmark (repetitions times ACDC size) usually overlap, demonstrating that the analysis time of M-GA depends mainly on the benchmark analyzed, and not on ACDC size. There is some variability in the execution time for the same benchmark (non-overlapping points) in only 11 out of 920 executions. The M-GA analysis time is less than a day for all except six of the benchmarks, with the longest analysis (that for *powerwindow*) taking around 5 days. Therefore, in general, benchmarks which take more time to analyze end up running alone for their last analyses, and hence, they share fewer hardware resources and show more variability in analysis time, as can be seen in Figure 6.

The most demanding operation in M-GA is *fitness*, i.e., performing the WCET analysis [16] to estimate the WCET bound for all individuals in each iteration of the genetic algorithm. The application of the other operators requires less than 2% of M-GA analysis time.

Note that, for simplicity, our M-GA implementation does not run in parallel. That is, all 99 individuals $\times$ 200 generations of each experiment are executed sequentially in a single core. With trivial parallelization, the times we present would be divided by the number of cores used. That is, *powerwindow* would take around 2 hours using our 55 cores. In any case, it is important to recall that the genetic algorithm is executed during the design stage of the system, and hence, the more solutions it explores at this time, the better the solutions it will find, and the better the system run-time execution will be.

D. WCET RESULTS

In this section, we show how the different configurations for the ACDC affect the WCET bound. Recall that the *rad2deg*, *cosf*, and *statemate* benchmarks have just one outer loop (see Figure 3), and hence, they are not suitable for multiple-content experiments.

Let us start with an overall summary comparing all the ACDC configuration methods studied. For each method, Table 7 shows its behavior with respect to the S-EB method, taken as the baseline. That is, for the ACDC configuration obtained for each experiment, it compares the WCET bound computed with this configuration with the WCET bound obtained using the S-EB configuration. Overall, the genetic algorithm methods improve a higher percentage of the experiments. Specifically, S-GA improves 56% of the experiments with respect to using S-EB. Since S-GA uses the configuration of S-EB as its initial solution, S-GA can never be worse than with S-EB, as indicated in Table 7. Similarly, M-GA provides better results for 54% of the experiments; however, 41% of its results are worse than with S-EB. As input, M-GA uses the solution provided by M-EB, which may be worse than those from S-EB, as shown in the table. Overall, at first glance, our proposed S-GA method is the best option. Note, however, that Table 7 shows the percentage of improved experiments, but not the extent of these improvements, which we discuss below.

Let us now provide a summary of the actual WCET improvements that each method achieves with respect to S-EB. Table 8 shows the main statistics for the WCET bounds obtained from the experiments run. Values are normalized with respect to the bound obtained with the S-EB method. That is, methods having values lower than 1 means that they reduce (improve on) the WCET bound compared to that obtained with S-EB. Recall that S-GA uses the S-EB solution as its initial solution, and hence, the value for S-GA in the *Maximum* column is 1. On average, S-GA, M-EB, and M-GA obtain the best WCET bounds, with 95%, 97%, and 95% reductions in WCET, respectively. The best WCET bound is obtained by M-EB and M-GA, with a value of 0.44, meaning that for a particular experiment, the WCET bound obtained with the ACDC configuration provided by both M-EB and M-GA is less than half of that obtained using the ACDC configuration provided by S-EB. Recall that M-GA uses the output of M-EB as a starting point to search for better solutions, and therefore, M-GA values are always better than or equal to those from the M-EB method. Considering the results in Table 8, at first glance, our proposed methods M-GA and M-EB are the best ones.

Both Table 7 and Table 8 show aggregated results, which provide an initial insight into the problem overall but may hide subtle trends. In order to comprehend the whole problem, a detailed representation and discussion of all the results is required. Figure 7 shows a comparison of the WCET bounds calculated for each ACDC configuration method. As above, the baseline configuration method is S-EB, with a relative WCET bound value of 1. The WCET bound of the other methods is presented with respect to this baseline. We should bear in mind that all these methods are heuristic, and hence, the optimal configuration (the one resulting in the shortest WCET) is unknown. The S-EB and S-OSP methods use different criteria to configure the ACDC: S-EB grants DRP to the load/store instructions which, in isolation, would obtain the greatest benefit in the estimated always-miss WCET, and S-OSP grants the DRPs that would be

TABLE 5. WCET bounds (average, minimum and maximum of ten repetitions, in cycles) obtained using M-GA as the ACDC configuration method.

	4 entries			8 entries			16 entries			32 entries		
Name	Average	Min.	Max.	Average	Min.	Max.	Average	Min.	Max.	Average	Min.	Max.
audiobeam	95972	95800	96164	93472	93195	93611	90692	90414	90986	88929	88365	89171
binarysearch	1330	1330	1330	1304	1304	1304	1304	1304	1304	1304	1304	1304
bsort	7403	7335	7429	7429	7429	7429	7373	7335	7429	7382	7335	7429
cjpeg_transupp	19174	18956	19268	17565	17565	17565	16226	16226	16226	15927	15927	15927
complex_updates	4319	4319	4319	2551	2551	2551	2083	2083	2083	2083	2083	2083
countnegative	1816	1816	1816	1790	1790	1790	1790	1790	1790	1790	1790	1790
fft	163978	163647	164310	124293	123763	124426	124360	123763	124426	124360	123763	124426
filterbank	12552	10505	12780	10876	8542	11883	9611	7874	10994	10404	8394	10994
fir2dim	3651	3625	3781	3708	3625	3781	3750	3625	3781	3751	3625	3781
g723_enc	107685	107685	107685	90109	90109	90109	79156	79156	79156	77665	77665	77665
gsm_dec	51514	50610	52209	48453	47230	50883	45693	43434	46619	42389	40655	42813
iir	1994	1994	1994	1890	1890	1890	1838	1838	1838	1838	1838	1838
insertsort	2094	2094	2094	2002	1990	2107	1951	1951	1951	1970	1951	1990
isqrt	4136	4118	4170	4102	4092	4131	4102	4092	4131	4102	4092	4131
jfdctint	4444	4444	4444	3820	3820	3820	3339	3339	3339	3339	3339	3339
lms	20049	20049	20049	19906	19906	19906	19906	19906	19906	19906	19906	19906
ludcmp	2594	2433	2681	2565	2447	2902	2578	2512	2694	2663	2538	2772
matrix1	1825	1824	1837	1806	1615	1837	1746	1654	1837	1778	1732	1824
minver	3943	3741	3989	3729	3690	3768	3642	3612	3664	3719	3612	3846
petrinet	17339	17326	17391	17130	17066	17144	16624	16624	16624	15584	15584	15584
powerwindow	11090655	10670322	12079430	9914176	9914176	9914176	9226484	9226484	9226484	8461818	8461818	8461818
rijndael_enc	1071960	1071960	1071960	1064026	1063984	1064400	1056720	1056405	1056834	1046751	1046421	1046850
st	54853	54853	54853	54644	54644	54644	54644	54644	54644	54644	54644	54644

TABLE 6. Statistics for min(*WCET*)/ max(*WCET*) for M-GA.

Items:	92
Average:	0.97
Minimum:	0.71
Maximum:	1.00
Standard Deviation:	0.06

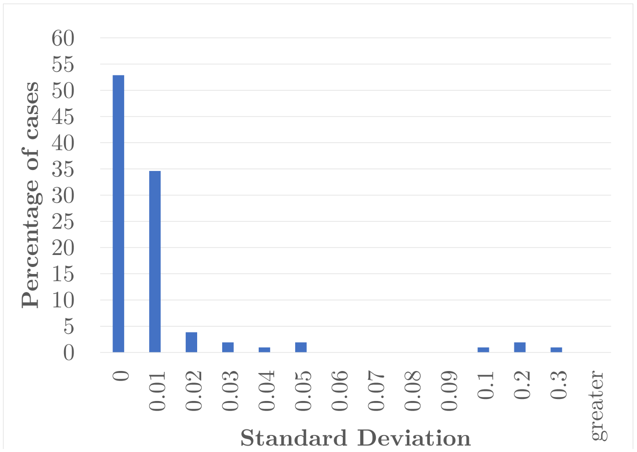

FIGURE 5. Histogram of the standard deviation (WCET obtained by M-GA) within the ten runs of each experiment ⟨benchmark, ACDC entries⟩.

TABLE 7. Percentage of experiments showing better results than those obtained with the S-EB method.

Method	Better	Worse	Tie
S-OSP	15%	57%	24%
S-GA	56%	—	44%
M-EB	35%	56%	8%
M-OSP	33%	63%	4%
M-GA	54%	41%	5%

optimal assuming that there is no interference between paths. In turn, S-GA applies a genetic algorithm using the S-EB configuration as the starting point. Apart from these single-content configuration methods (i.e., those that specify a single configuration for the whole task execution), we also test our proposed multiple-content configuration versions (i.e., those that may change the ACDC configuration during run-time before outer loops). To visualize this more clearly, in Figure 7, the ACDC configuration methods tested (on the x-axis) are also indicated with different shapes and colors. The number of ACDC entries (4 to 32 entries, 256 B to 2 KiB of data cache size) can be seen on the labels on the right, whereas the different benchmarks (ordered by their number of static load/store instructions, see Figure 3) are labeled at top of the figure. The rightmost subplot shows the average results considering all benchmarks.

As can be seen, results are fairly stable around 1 (the S-EB method baseline). Further, there is no clear relationship between ACDC size and WCET reduction. S-GA and S-OSP show the most direct relationship, with greater improvements being seen with larger ACDC sizes in 10 and 6 out of

TABLE 8. Main statistics for the WCET bounds of the studied methods. Values are normalized taking the WCET bound of S-EB as baseline (method / S-EB), i.e., the lower, the better.

Method	Maximum	Minimum	Average	Std. Deviation
S-OSP	4.89	0.64	1.17	0.51
S-GA	1.00	0.51	0.95	0.09
M-EB	1.24	0.44	0.97	0.14
M-OSP	2.49	0.51	1.07	0.27
M-GA	1.12	0.44	0.95	0.13

26 benchmarks respectively. On the other hand, for M-GA, the opposite trend is seen with more benchmarks: greater improvement being seen with larger ACDC size in only 4 out of 23 benchmarks, while greater improvement is seen with smaller ACDC size in 8 benchmarks.

Multiple-content configurations provide better results for the smallest ACDC (4 entries), since with its capacity only four loads/stores can be granted replacement permission. That is, some benchmarks improve their worst-case performance if the four instructions selected are changed at different execution points in the program (before outer loops). This is the case, for instance (Figure 7), for *matrix1*, *iir*, *fir2dim*, *complex_updates*, *fft*, *st*, and *cjpeg_transupp*, where the multiple-content configurations clearly outperform single-content ones for an ACDC with just 4 entries. When the ACDC size grows, it can accommodate the required replacement permissions with a single configuration, and therefore, dynamically reconfiguring the ACDC provides less benefit or none at all. The number of benchmarks clearly taking advantage of reconfiguration falls from these 7 benchmarks (out of 26) for an ACDC with 4 entries to 4 benchmarks for an ACDC with 8 entries, and for an ACDC with 32 entries, only *audiobeam* and *powerwindow* seem to still marginally benefit from ACDC reconfiguration.

Considering the results of each method, the most extreme cases are found with S-OSP. That is, its heuristic criteria obtain the worst results for some benchmarks (namely, *bsort*, *matrix1* for an ADCS with 4 entries, *countnegative*, *lms*, *fft*, *g723_enc*, *gsm_dec*, *cjpeg_transupp*, and *rijndael_enc*), although they work very well for others (e.g., *cosf* for any ACDC size, and *filterbank* for ACDCs with 16 and 32 entries). Indeed, the WCET results of S-OSP for *cjpeg_transupp* for ACDCs with 8 and 16 entries are around 3.5 and 5 times worse than those of S-EB, respectively, outside the range of the area plotted.

Finally, S-GA obtains very good results in general, since it performs an exploration of possible configurations, and uses S-EB as the starting point. It has a drawback, however, namely, the time required to perform this exploration, as described in Section VI-C. Nevertheless, bearing in mind that a real-time system is not usually modified once designed, performing a thorough exploration of its configuration is still advisable.

Figure 8 shows the results presented above from a different point of view. In this case, we compare the methods tested

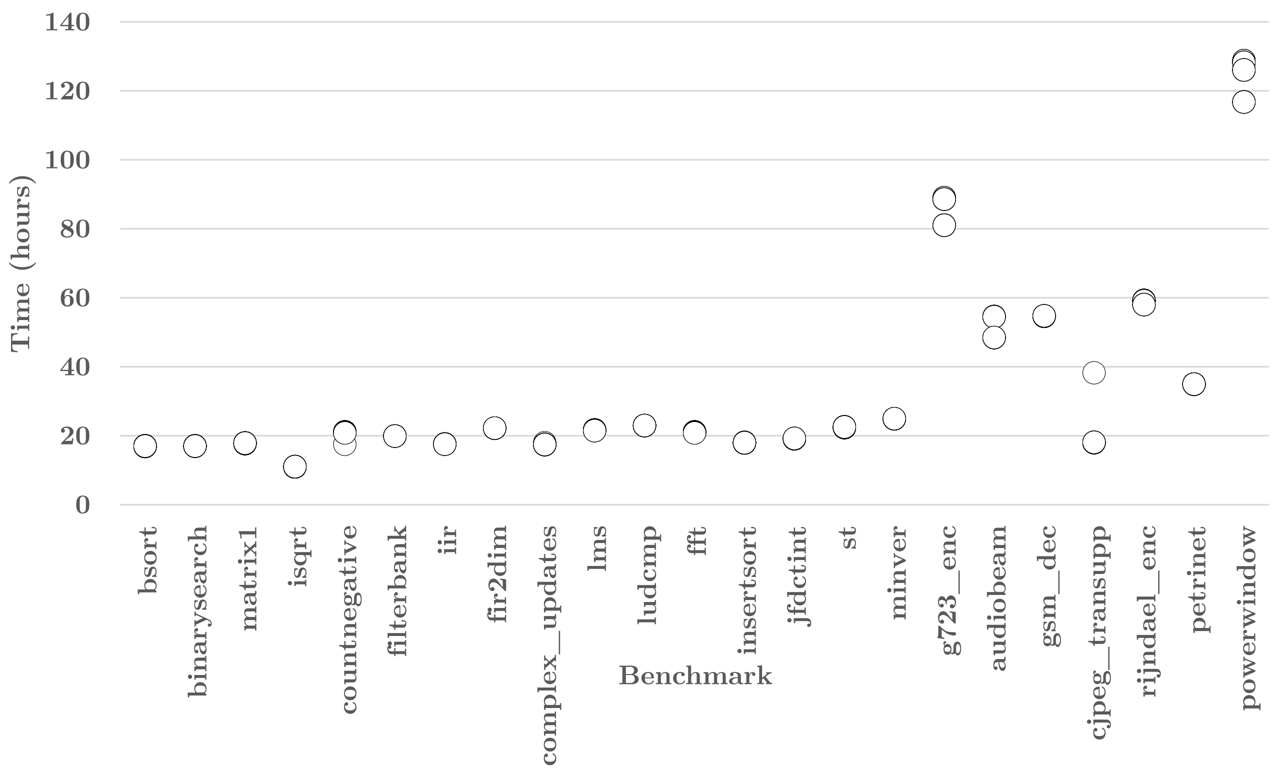

FIGURE 6. **Scatter plot of M-GA execution time versus benchmark. There are 40 points per benchmark (most of them overlapping) corresponding to ten runs times four ACDC sizes per benchmark.**

showing which one provides the best results for each ACDC size, and how much better they are with respect to those with other sizes. The WCET bound obtained is plotted on the y-axis, relative to the best bound for an ACDC with 4 entries, and the ACDC sizes tested, for each benchmark, on the x-axis. There are many situations in which several methods obtain the same ACDC configuration, and thus provide the same WCET bound. For such cases, we rank the methods in the following order: S-EB, S-OSP, S-GA, M-EB, M-OSP, and M-GA. For instance, if both S-EB and S-GA provide the best result for a given case, S-EB is shown as the best choice. Our order of preference is based on the following reasoning. Multiple-content configuration methods are more complex than single-content methods, and hence, are less desirable. Genetic algorithms require much more computation time, so they are considered the last resort, both for single and multiple-content methods. Finally, S-EB is based on the standard IPET model for WCET computation and is relatively easy to integrate into any tool, whereas S-OSP requires the more restricted LockMS model, and therefore, S-EB is preferable to S-OSP.

Note that the values shown for both S-GA and M-GA are the average of those found by the corresponding genetic algorithm, as explained above. Seeking to gain insight into the potential WCET improvement, we also show the best result found by our genetic algorithms as black horizontal lines. It can be seen, for instance, that particular genetic algoritm runs of *filterbank* and *cosf* obtain much better results than their corresponding average.

As expected, the larger the ACDC, the better the WCET bound in general. However, note that the ACDC exploits existing data reuse, like any other data cache. Therefore, if this reuse can be completely exploited by a small ACDC, a larger one will not achieve any improvement. This can be seen in our smallest benchmarks (on the left of Figure 8). In these cases, at certain ACDC sizes, the WCET no longer decreases. On the other hand, our largest benchmarks (on the right of Figure 8) would benefit from even larger ACDCs, since their resulting WCET bound decreases linearly with increasing ACDC size. It is also important to note that the ACDC has a fully-associative design, and hence, its energy consumption is higher than that of direct-mapped caches. Although the associative level tested is relatively small, ACDC designs with more entries may have energy consumption issues.

To help interpret the patterns in Figure 8, it is useful to plot the results in a different way. Figure 9 shows the number of occurrences of each method for each ACDC size in Figure 8. In this graph, several patterns emerge that were not evident in the previous figure. As observed above, multiple-content methods (M-EB, M-OSP, and M-GA) work better for very small ACDCs. Thus, the larger the ACDC, the fewer times such methods provide the best result, especially in the case of M-EB, which shows a clear downward pattern.

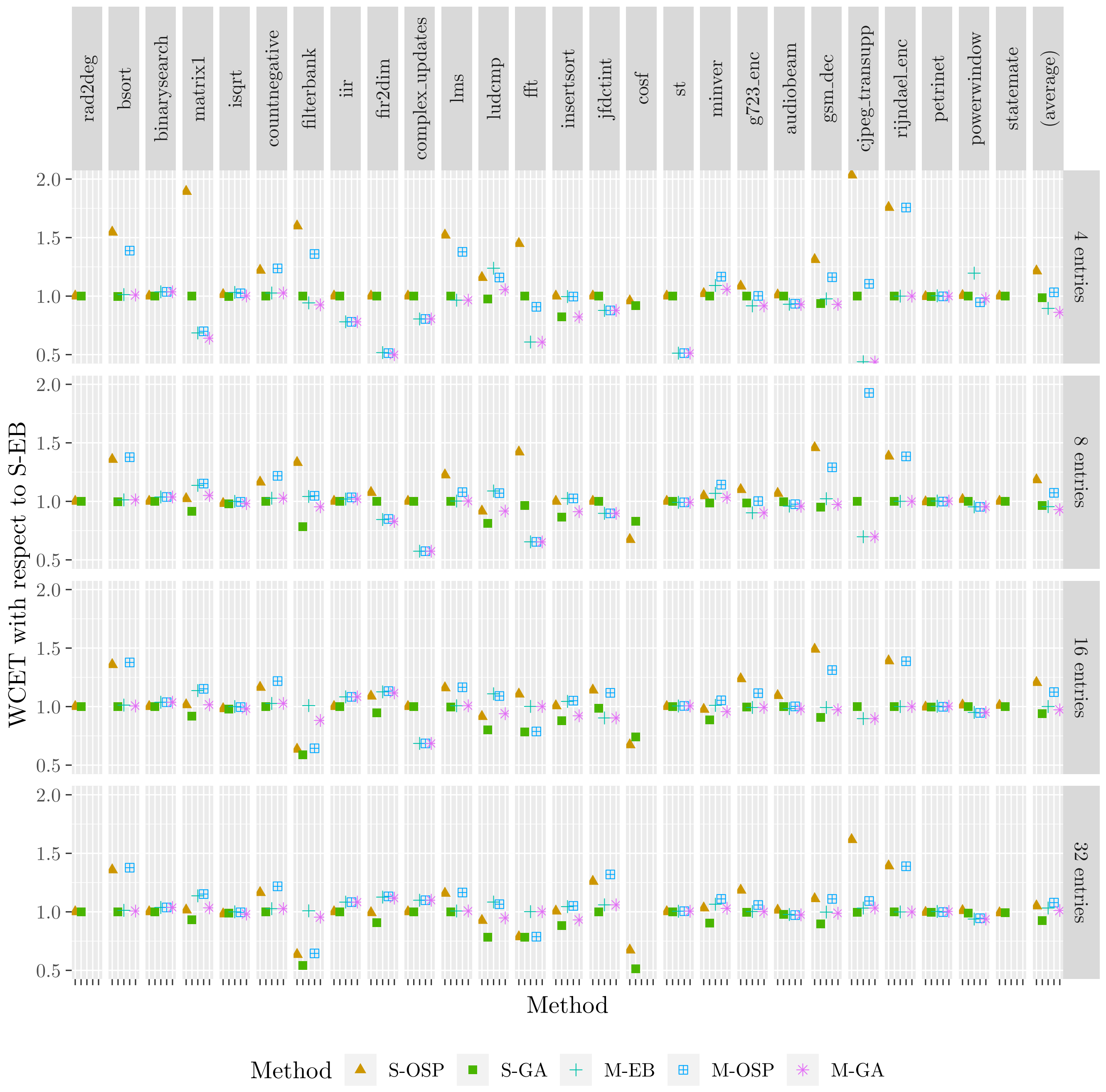

FIGURE 7. Relative WCET bounds.

Focusing on the single-content methods (S-EB, S-OSP, and S-GA), S-GA shows better results as the number of entries in the ACDC grows. In our tests, S-EB is always better than S-OSP, and S-GA provides the best results for ACDCs with 8 or more entries. This behavior can be easily explained by taking into account that the more ACDC entries, the larger the configuration space, that is, there are many more possible configurations. In such cases, a genetic algorithm shows its capacity to obtain a good ACDC configuration.

Based on our results, we propose the following guidelines for configuring an ACDC. If the number of entries in the ACDC is very small, S-EB or its multiple-content version M-EB should provide good results, though M-GA might improve on them. If the number of entries in the ACDC is moderate or large, S-EB should offer a good ACDC configuration, but applying S-GA is likely to improve on the results. For large ACDCs, testing for multiple-content configurations would not be required.

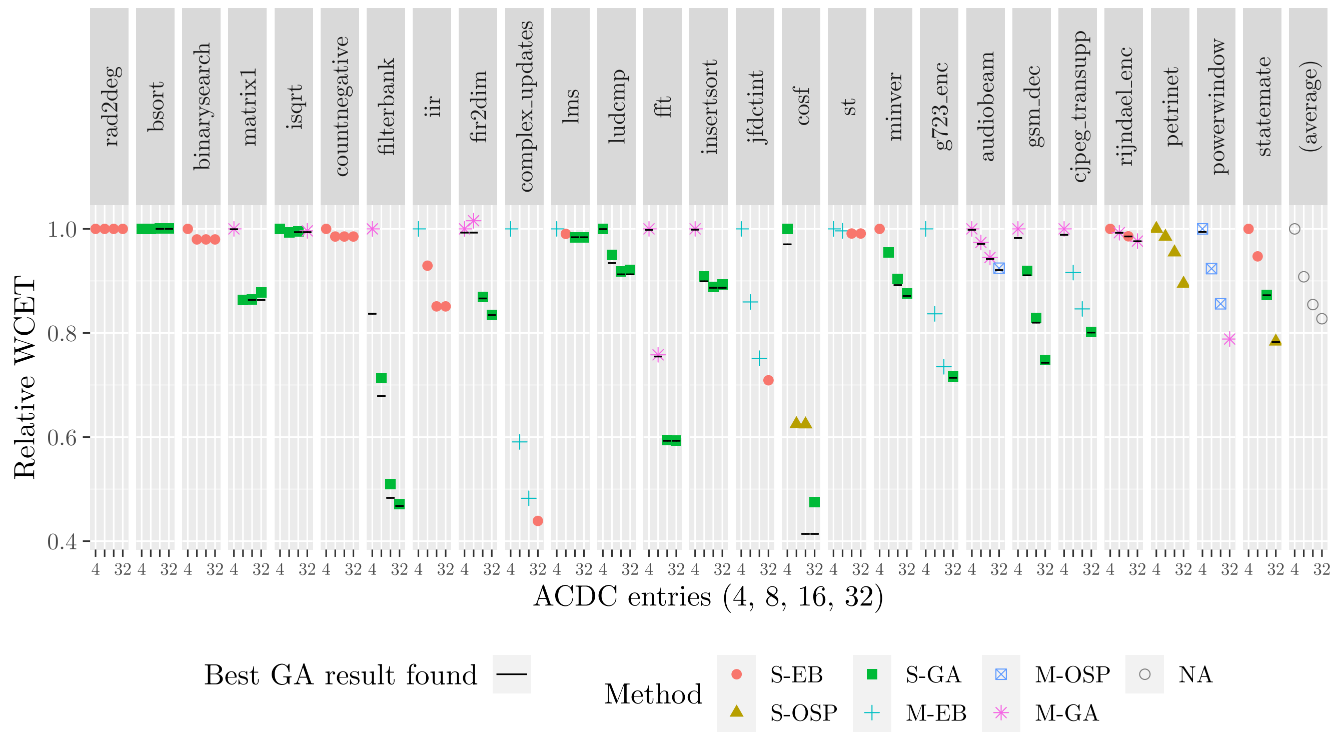

FIGURE 8. Best methods for obtaining the best WCET bounds, depending on the number of entries in the ACDC. Relative WCETs (y-axis) show the WCET bound obtained with respect to the best result for an ACDC with 4 entries.

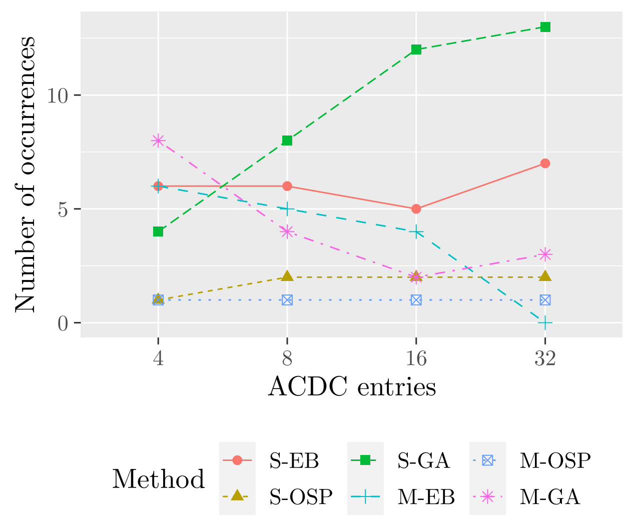

FIGURE 9. Number of times that a given method is the best for a given ACDC size, in the benchmarks tested.

VII. CONCLUSION

The ACDC is a small predictable data cache, specially designed for real-time systems. It works by granting data replacement permission to a set of preconfigured load/store instructions. It is, however, hard to know how to select which loads/stores should be granted this permission to improve the WCET. In this paper, we propose and study several different methods to configure the ACDC in order to minimize the resulting WCET bound. We start with two existing heuristic methods (herein called S-EB and S-OSP). These methods assume single-content configurations, that is, they provide a single configuration to be used during the whole execution of a task. Additionally, to compare with these two methods, we propose a genetic algorithm to test whether the existing solutions are competitive, and explore whether a genetic algorithm might improve on them. Further, we propose three multiple-content configuration methods, that is, methods that provide different configurations to be used in different execution phases of a task. Our multiple-content configuration methods are based on the same heuristics as the aforementioned single-content ones.

Our results show that the new proposed methods reduce the WCET by 5% in average, and up to 50% in some cases. The single-content method S-EB is the most stable, i.e., it provides good configurations in most benchmarks, for any ACDC size. This means that, for benchmarks larger than those considered here, this method would be a robust candidate to test. On the other hand, M-EB obtains its best results for very small ACDCs, that is, when using multiple-content configurations is a decisive factor. Although there are few such cases in TACLeBench, this might be the general situation for real codes. That is, the multiple-content method we propose, M-EB, is likely to generally outperform S-EB in larger tasks. Regarding our genetic algorithm proposals, they obtain the best results in many cases. In general, S-GA obtains the best results for medium-to-large ACDCs; that is, when the solution space is very large. In contrast, M-GA obtains the best results when the ACDC size does not suffice for single-content configurations. On the other hand, while the cost of running the genetic operators is very low, the cost of evaluating the WCET bound, that is, computing the fitness of each individual in all generations, makes the genetic algorithms computationally costly. Despite this, bearing in mind that minimizing the WCET bound is a critical part of any real-time system, we recommend running these algorithms, either to confirm that the starting solution is good, or to find a better one.

REFERENCES

[1] 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.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[3] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Design Autom. Electron. Syst.*, vol. 21, no. 3, pp. 1–24, Jul. 2016.

[4] 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.

[5] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 2003, pp. 272–282.

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

[7] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for WCET analysis of hard real-time multicore systems," in *Proc. 36th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 57–68.

[8] A. M. Campoy, P. Ivars, and J. V. B. Mataix, "Static use of locking caches in multitask preemptive real-time systems," in *Proc. IEEE Real-Time Embedded Syst. Workshop*, Dec. 2001, pp. 1–6.

[9] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23rd IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2002, pp. 114–123.

[10] A. M. Campoy, A. Perles, F. Rodriguez, and J. V. Busquets-Mataix, "Static use of locking caches vs. dynamic use of locking caches for real-time systems," in *Proc. Can. Conf. Electr. Comput. Eng. Toward Caring Humane Technol. (CCECE)*, May 2003, pp. 1283–1286.

[11] 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.

[12] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Proc. 18th Euromicro Conf. Real-Time Syst. (ECRTS)*, 2006, pp. 217–226.

[13] L. C. Aparicio, J. Segarra, C. Rodriguez, and V. Vinals, "Combining prefetch with instruction cache locking in multitasking real-time systems," in *Proc. IEEE 16th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2010, pp. 319–328.

[14] 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.*, vol. 61, nos. 5–6, pp. 227–238, May 2015.

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

[16] J. Segarra, R. Gran Tejero, and V. Viñals, "A generic framework to integrate data caches in the WCET analysis of real-time systems," *J. Syst. Archit.*, vol. 120, Nov. 2021, Art. no. 102304.

[17] J. Segarra, J. Cortadella, R. G. Tejero, and V. Vinals-Yufera, "Automatic safe data reuse detection for the WCET analysis of systems with data caches," *IEEE Access*, vol. 8, pp. 192379–192392, 2020.

[18] 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.

[19] E. D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Reading, MA, USA: Addison-Wesley, 1989.

[20] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," in *Proc. 14th Int. Conf. Real-Time Netw. Syst. (RNTS)*, Poitiers, France, May 2006, pp. 1–10.

[21] A. M. Campoy, E. Tamura, S. Sáez, F. Rodríguez, and J. V. B. Mataix, "On using locking caches in embedded real-time systems," in *Proc. ICESS*, Dec. 2005, pp. 150–159.

[22] Inc. Micron Technology. *Automotive DDR SDRAM MT46V32M8, MT46V16M16*. Accessed: Oct. 14, 2021. [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobiledram/low-power-dram/lpddr/256mb_x8x16_at_ddr_t66a.pdf

[23] 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.

[24] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, 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.*, vol. 55. Toulouse, France, Jul. 2016, pp. 2:1–2:10.

[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] 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.

JUAN SEGARRA received the graduate degree in computer science and the Ph.D. degree from Universitat Jaume I, Spain, in 1999 and 2003, respectively. In 2003, he joined the University of Zaragoza, where he is currently working as a Lecturer with the Informática e Ingeniería de Sistemas Department. He is a member of the Computer Architecture Group at the University of Zaragoza (gaZ). His research interests include QoS, media distribution, worst-case execution time, and worst-case memory performance in hard real-time systems.

ANTONIO MARTÍ-CAMPOY was born in Valencia, Spain, in 1969. He received the M.Sc. and Ph.D. degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1996 and 2003, respectively. In 1996, he joined the Department of Computer Engineering at the UPV, where he is currently an Associate Professor and a member with the Institute for Advanced Information and Communication Technologies Applications (ITACA). His research interests include real-time systems, embedded systems, and active learning methodologies.