ACDC: Small, Predictable and High-Performance Data Cache

JUAN SEGARRA, Universidad de Zaragoza
CLEMENTE RODRÍGUEZ, Universidad del País Vasco
RUBÉN GRAN, LUIS C. APARICIO, and VÍCTOR VIÑALS, Universidad de Zaragoza

In multitasking real-time systems, the worst-case execution time (WCET) of each task and also the effects of interferences between tasks in the worst-case scenario need to be calculated. This is especially complex in the presence of data caches. In this article, we propose a small instruction-driven data cache (256 bytes) that effectively exploits locality. It works by preselecting a subset of memory instructions that will have data cache replacement permission. Selection of such instructions is based on data reuse theory. Since each selected memory instruction replaces its own data cache line, it prevents pollution and performance in tasks becomes independent of the size of the associated data structures. We have modeled several memory configurations using the Lock-MS WCET analysis method. Our results show that, on average, our data cache effectively services 88% of program data of the tested benchmarks. Such results double the worst-case performance of our tested multitasking experiments. In addition, in the worst case, they reach between 75% and 89% of the ideal case of always hitting in instruction and data caches. As well, we show that using partitioning on our proposed hardware only provides marginal benefits in worst-case performance, so using partitioning is discouraged. Finally, we study the viability of our proposal in the MiBench application suite by characterizing its data reuse, achieving hit ratios beyond 90% in most programs.

Categories and Subject Descriptors: B. [**Hardware**]; B.3 [**Memory Structures**]; B.3.2 [**Design Styles**]: *Cache memories*; C. [**Computer Systems Organization**]; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Design, Performance

Additional Key Words and Phrases: Worst-case analysis

ACM Reference Format:
Juan Segarra, Clemente Rodríguez, Rubén Gran, Luis C. Aparicio, and Víctor Viñals. 2015. ACDC: Small, predictable and high-performance data cache. ACM Trans. Embedd. Comput. Syst. 14, 2, Article 38 (February 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2677093

This work was supported in part by grants TIN2007-60625 and TIN2010-21291-C02-01 (Spanish government and European ERDF), gaZ: T48 Research Group (Aragón government and European ESF), Consolider CSD2007-00050 (Spanish government), and HiPEAC-2 NoE (European FP7/ICT 217068).

Authors' addresses: J. Segarra, R. Gran, L. C. Aparicio, and V. Viñals, Universidad de Zaragoza, Edificio Ada Byron, C/ María de Luna, 1, 50018 Zaragoza, (SPAIN); emails: {jsegarra, rgran, luisapa, victor}@unizar.es; C. Rodríguez, Universidad del País Vasco, Manuel Lardizabal, 1, 20018 Donostia-San Sebastián, (SPAIN); email: acprolac@ehu.es.

DOI: http://dx.doi.org/10.1145/2677093

1. INTRODUCTION

Real-time systems require that tasks complete their execution before specific deadlines. Given hardware components with a fixed latency, the worst-case execution time (WCET) of a single task could be calculated from the partial WCET of each basic block of the task. However, to improve performance, current processors perform many operations with a variable duration. A memory hierarchy made up of one or more cache levels takes advantage of program locality and saves execution time and energy consumption by delivering data and instructions with an average latency of a few processor cycles. Unfortunately, the cache behavior depends on past references, and generally it is necessary to know the previous access sequence to calculate the latency of a given access in advance. Resolving these *intratask* interferences is a difficult problem in its own right. Moreover, real-time systems usually work with several tasks that may interrupt each other at any time. This makes the problem much more complex, since the cost of *intertask* interferences must also be identified and bounded. Furthermore, both of these problems cannot be accurately solved independently, since the path that leads to the worst case of an isolated task may change when considering interferences.

In this article, we propose a small data cache that effectively exploits locality. Instead of a conventional data-driven data cache, we propose an instruction-driven data cache, where selected memory instructions are associated with particular data cache line frames. These data cache line frames can only be replaced by their associated instructions—that is, only such instructions have data cache replacement permission. Since each memory instruction replaces its own data cache line frame, it prevents pollution (i.e., conflicts that evict highly reusable content), and its performance is independent of the size of the data structures in tasks. Assuming that all instructions have data cache replacement permission, the number of hits and misses in our proposed data cache can be estimated using the data reuse theory [Wolf and Lam 1991]. Next, any WCET optimization method can be used to decide which instructions have such permissions, depending on the cache size, the intertask interferences, and so forth. To obtain such instructions, we extend the Lock-MS WCET analysis method [Aparicio et al. 2010, 2011].

Compared to a conventional data cache, the novel features of our data cache design are the following:

—*High-performance achievement*: Contents are replaced in a similar way to in conventional caches, maintaining its dynamic behavior.
—*Predictability and no pollution*: Only selected instructions can replace data cache lines. This is achieved by static indexed-based replacement, with the advantage that the usual replacement overhead (e.g., control and state bits of LRU) is eliminated.
—*Small cache size (256 bytes), independent of the size of the data structures in tasks*: Each data structure requires a single data cache line at most, so in many cases, even with this small size, several cache lines are left unused.
—*The number of misses is estimated using data reuse theory as developed for conventional caches* [Wolf and Lam 1991]: This enables references to unknown addresses (e.g., pointers) to be analyzed, which is not possible with other methods (e.g., Li et al. [1996]; White et al. [1997]).

Compared to scratchpad memories, the performance of our proposed design does not depend on program data size, it has no specific problems when analyzing pointers, and it does not require data addresses to be tuned [Whitham and Audsley 2010].

The rest of this article is organized as follows. Section 2 reports related background on caches. Our proposed data cache is described in Section 3. Section 4 shows how to specify its behavior as an ILP model, including details of how the number of misses can be estimated by applying the data reuse theory. Sections 5 and 6 describe the

real-time experimentation environment and the worst-case performance results obtained. Section 7 extends experiments to larger programs without real-time requirements. Finally, Section 8 presents our conclusions.

2. RELATED WORK

As outlined previously, caching is a difficult problem in real-time systems. Many studies have been reported on instruction caches and their analysis. They can be divided into those that analyze the normal behavior of the instruction cache and those that restrict its behavior to simplify the analysis. The first approach involves attempting to model each task and system as accurately as possible considering the dynamic behavior of the cache [Li et al. 1996; White et al. 1997; Lundqvist and Stenström 1999; Theiling et al. 2000; Aparicio et al. 2008]. Since this analysis is complex, interferences between tasks are not usually considered, with tasks being analyzed in isolation. This means that complementary methods (analysis of cache-related preemption delays) are needed to adapt the WCET of each isolated task to multitasking systems (e.g. Altmeyer et al. [2010]). On the other hand, cache-locking techniques restrict instruction cache behavior, disabling cache replacement, a possibility offered by many commercial processors [Martí Campoy et al. 2001; Puaut and Decotigny 2002; Suhendra and Mitra 2008]. With specific contents locked in cache, the timing calculations are easier, so these methods can enable the full system to be analyzed (i.e., several tasks on a real-time scheduler). Cache-locking techniques can also be divided into *static* and *dynamic cache locking*. Static locking methods preload the cache content at system start-up and fix this content for the whole system lifetime so that it is never replaced [Martí Campoy et al. 2001; Puaut and Decotigny 2002]. Dynamic cache locking, on the other hand, allows tasks to disable and enable the cache replacement. Although there are studies that allow instruction cache reloading at any point [Puaut 2006], most restrict reloading to context switches [Martí Campoy et al. 2003a, 2003b; Aparicio et al. 2010, 2011]. These approaches require per-task selection of contents, with the drawback that preloading is performed every time a task starts/resumes its execution. Locking instructions in a cache can be complemented with a line buffer, which effectively captures spatial locality [Puaut and Decotigny 2002; Aparicio et al. 2011]. Instruction prefetch components can also improve performance [Aparicio et al. 2010]. Further, to avoid interferences between tasks in real-time multitasking systems, cache partitioning may be used [Reddy and Petrov 2007].

Data caching is much more complex than instruction caching, as references may present very different behaviors: scalar versus nonscalar, global versus local, stack frame (i.e., subroutine context), dynamic memory, and so forth. Most proposals use the memory model of C: local and temporary variables and parameters stored on the *stack*, global variables and (some) constants in the global *static* data region, and dynamic variables on the *heap*. Thus, instead of a single component (data cache) exploiting them all, some approaches specialize in exploiting particular access patterns in separate caching structures. One of the most straightforward specialization is exploiting spatial and temporal locality into two separate caches [González et al. 1995]. The instruction address of the memory access (load/store) and a hardware predictor allow prediction of the type of locality. The size of such caches is higher than 8KB, and they have no pollution for references to nonscalar variables. Early approaches focused on a stack cache [Ward and Halstead 2002]. Other authors have suggested hardware modification to include a register-based structure for storing part of the stack frame [Gonzalez-Alberquilla et al. 2010]. There are also proposals to store accesses to the heap in a small cache (2KB) or a large cache (32KB) [Geiger et al. 2005]. Finally, one study proposed three caches to manage the three memory regions (stack, global, and heap) [Lee and Tyson 2000]. Additionally, this avoids conflicts between regions and provides the required size for each: small for the stack and global, and large for the heap.

All previous techniques have a low energy consumption due to their small size. They do, however, suffer from the problem of pollution to some extent, especially when accessing local structures with spatial locality. Their results show marked variations depending on the size of such structures.

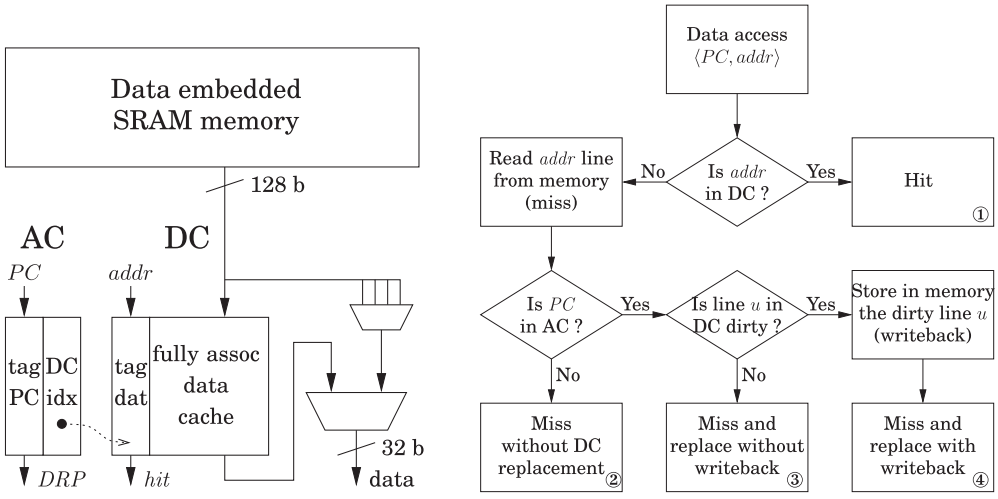
It is worth mentioning the proposal of Tyson et al. [1995], which has some similarities to our proposal. They suggest an instruction-driven, selective allocation policy for the data cache. After a miss, allocation is allowed or not by the memory instruction causing it, either statically or dynamically. The static approach selects instructions after an application profiling of individual miss ratios and requires an extended instruction set, whereas the dynamic approach relies on a hardware prediction table that records the hit/miss behavior of each load instruction. However, they focus on improving statistical performance using large structures and do not associate cache lines with memory instructions as in our replacement policy.

Locking data caches and scratchpad memories are alternatives intended to capture temporal locality and avoid pollution in real-time systems [Puaut and Decotigny 2002; Puaut and Pais 2007]. However, exploiting spatial locality is still a problem. Since different data structures may be used in different parts of a task and they may be too large to fit into a locked data cache, some authors propose a dynamic locking mechanism where tasks include code to lock/unlock the data cache, and also to preload its contents at runtime [Vera et al. 2003; Xue and Vera 2004; Vera et al. 2007; Whitham and Audsley 2010]. The selection of data memory lines (or scratchpad content) is based on estimations of the number of misses for different chunks of code. The number of misses can be predicted using cache miss equations [Ghosh et al. 1999], based on the data reuse theory for LRU replacement in conventional data caches [Wolf and Lam 1991]. Therefore, if preloading and locking the data cache with a given selection of data reduces the number of misses in the worst case, the required preload/lock/unlock instructions are inserted into the code. In general, whole data structures are preloaded to guarantee hits if they fit in the cache. Otherwise, the data cache may also be locked to reduce pollution. This technique is particularly sensitive to optimizations that increase locality (*padding* and *tiling*), as it is very dependent on the size of the referenced data structures. In addition, it is usually combined with partitioning techniques to avoid intertask interferences (e.g., Reddy and Petrov [2007]). Similarly to the cache miss equations [Ghosh et al. 1999], in this article we use a specialized version of the reuse theory to estimate the number of misses. However, our proposed hardware allows us to perform whole program analysis, whereas cache miss equations on conventional data caches are limited to perfectly nested loops without conditional expressions.

Our proposal does not lock specific data but dynamically caches the data used by selected instructions. This avoids pollution; performance is independent of the data size and allows the analysis of references to unknown addresses based on their reuse. Further, being a single hardware component, it is more efficient than structures specialized on different access patterns. Morever, modification of task code is not required, and compiler optimizations are not so important. Last, with our proposal, the cache-related preemption delay of any task is constant (based on its setup) and independent of the rest of the tasks.

3. ACDC STRUCTURE

Our proposed data cache is able to take advantage of temporal and spatial locality. Usually, data caches are data driven—that is, their behavior (and thus their WCET analysis) is based on *which* data addresses are requested and their request order. Our proposed data cache is instruction driven, which means that its behavior depends on the instructions accessing the data—that is, on *how* the data are accessed.



(a) Data memory architecture (b) Data access procedure

Fig. 1. Data memory architecture and access procedure. AC contains addresses of instructions with data cache replacement permission and the index to their associated DC line.

3.1. Hardware Description

Our proposed data cache structure, Address-Cache/Data-Cache (ACDC), is a small fully associative data cache with a static, instruction-driven, replacement policy. We assume a write-back policy, as it involves less traffic than write-through. Since the replacement policy is external to the DC, no replacement mechanism, such as LRU or PLRU, is needed, unlike in conventional caches. Thus, on read hits it provides the data, and on write hits it updates the data. On cache misses, it is checked whether the accessing instruction, either a load or a store instruction, has data replacement permission. If so, its associated data cache line is replaced with the missing data from main memory. Otherwise, the data cache is not modified, and the missing data is read/written from/to memory. Since each instruction with replacement permission is only allowed to replace a predefined data cache line frame (simply "cache line" from now on) for its references, there is no pollution.

This behavior can be implemented in many ways. We describe an implementation that does not modify the instruction set and relies on two cooperating structures, the Address Cache (AC) and the Data Cache (DC), in Figure 1(a). The DC is a small data cache supporting fully associative read/write word lookup and direct block write, which is indexed by data addresses (*addr* in Figure 1(a)). Hence, the DC could be a conventional fully associative data cache with its automatic replacement circuitry disabled. The AC is a table whose entries keep the instruction addresses (*tag-PC* field) that can replace DC lines and the specific DC lines they can replace (*DC-idx* field). Each instruction in the AC is associated to exactly one DC line. We assume an AC with fully associative lookup of the *tag-PC* field (*PC* indexing in Figure 1(a)) and direct field writing. During the execution of a given task, no replacement is needed, so once written, the AC can be viewed as a locked cache.

In context switches, the task starting/resuming the execution stores in the AC its own set of instruction addresses with replacement permission along with the index of the DC line they are allowed to replace. Then, data accesses are made as depicted in Figure 1(b). The DC is accessed with the desired data address (*addr*). On hits, the DC

reads or writes the requested data and the access completes (box ①). On misses, data in main memory are requested, and it must be determined whether the requested data line must replace a currently stored memory line in the DC or not. With our proposed implementation, the AC is accessed with the current memory instruction address (PC). On an AC miss, the DC will not be replaced (box ②). On AC hits, the data access has replacement permission (*DRP*), and the DC line to replace is indicated by the index stored in the AC (*DC-idx*). In this case, having a write-back policy, if the currently stored victim line u is dirty, it will be written back to memory (box ④). Otherwise, the DC will be refilled without writing back (box ③). It is important to note that since the DC is the first component to be looked up, replacements can only occur if the accessed data are not cached, so it is not possible to duplicate lines in the DC.

As outlined previously, the AC permission table can be implemented in many ways. For instance, it could be completely removed by enhancing the instruction set with specific load/store instructions with replacement permission on particular DC lines. Alternatively, the permission table could be implemented in the DC by adding to each cache line a list of instruction addresses allowed to replace this cache line. Although implementation alternatives and their performance are not studied in this paper, note that the implementation overhead of the described AC component is very small compared to a conventional cache. In our experiments, each AC entry stores 4 bits, whereas each entry in DC (or in a conventional cache) has between 128 and 512 bits. Even considering the corresponding tag space in AC, its area would hardly suffice to add more capacity to a conventional cache (e.g., having an addressable space of 1GB and memory lines of 64 bytes, the whole AC would require 512 bits, whereas a single DC entry would be larger than 540 bits).

Once understood, the operation of ACDC and reviewing its associativity features may be interesting. Associativity in a conventional cache determines two quantities that match: first in how many places a line must be looked up, and second in how many places a line can be placed. Let us consider now the ACDC. From a search standpoint, DC is clearly fully associative. However, from a placement standpoint (after miss), we can speak of a direct mapping between a high-level variable or data structure and one or more entries in the DC (through one or more AC entries).

3.2. Locality Exploitation

In our proposed ACDC, only specific memory instructions are allowed to replace a given cache line, each of them having a predefined DC line to work with. This allows WCET minimization/analysis to be carried out relatively easily, without limiting the benefits coming from spatial and temporal reuse. Although there are many situations in which such reuse can be exploited, in this article we consider only the most common cases. Such cases cover most situations and can be modeled in a simple way. In addition, although our actual analysis is performed on binary code, for clarity let us use source code to explain how reuse would be exploited.

Temporal reuse. Memory references present temporal reuse when they access the same memory address repeatedly during program execution. In this article, only accesses to scalar variables are considered for temporal reuse. Although array reuse generally is better described by spatial reuse (see later), temporal reuse in very small arrays can be exploited if they are managed as a very small series of scalar variables. When scalar variables are accessed, if the same memory line has been accessed before only by the same instruction, there is what can be referred to as *self-temporal* reuse [Wolf and Lam 1991] of the reference in this instruction. Otherwise, if the same line has been accessed before but by another instruction, this is described as *group-temporal* reuse. Figure 2 shows an example of the temporal reuse of different structures, namely global (static) variables, local variables (within the function scope), and function

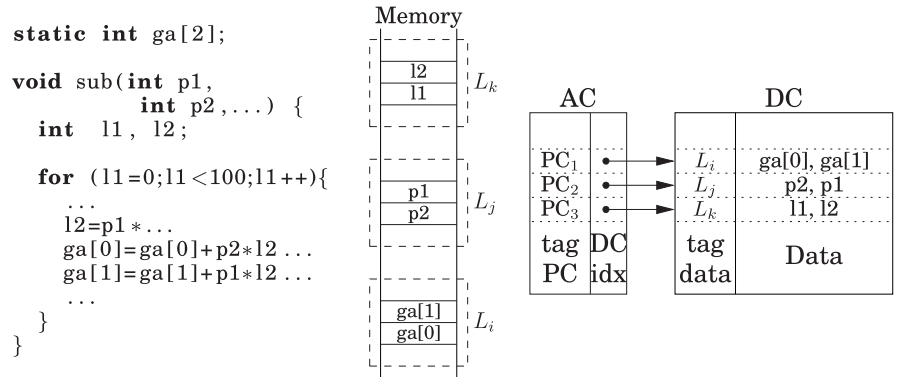


Fig. 2. Example of temporal locality in code, memory layout, and the ACDC structure. PC_1, PC_2, and PC_3 are the program counters of the first load/store instructions accessing the lines L containing the reused variables.

parameters. For variables not stored in registers, their location in memory (in specific memory lines L) will determine their associated temporal reuse. To exploit such reuse, the first reference to each of these memory lines will be given replacement permission so that they can be cached—that is, its PC will be included in the AC with an associated DC line to cache this data line. For instance, there will be only a single miss for references to the global small array variable ga (the first access), as all references to this variable will access the same memory line. Furthermore, assuming that the function sub always works with the same stack addresses (roughly speaking, it is always called from the same nesting level), all of its accesses will have temporal locality, with a single miss on the first access to each memory line. If sub is called from n different stack addresses, there will be n misses for each line with temporal locality instead of a single one. This would be the behavior for a task being executed without context switches. If context switches are allowed, as in our experiments that follow, each context switch will require preloading the AC and also saving and restoring the DC content. As we show later, even with a high number of context switches, these costs are relatively small due to the small size of our proposed structure.

Spatial reuse. There is spatial reuse of memory references when they access close memory addresses during program execution [Wolf and Lam 1991]. In this article, only sequential accesses (those with *stride* 1 element) are considered (independently of the size of this element), because other strides are unusual in real-time applications and formulas would become less clear. Nevertheless, considering other constant strides would be trivial. As earlier, if a given case of spatial reuse involves a single instruction, it is considered to be *self-spatial* reuse, whereas it is referred to as *group-spatial* reuse when several instructions are involved.

Let us illustrate the spatial locality in matrix multiplication codes (Figure 3) using our proposed ACDC structure. We present three cases, namely the matrix multiplication code of the *matmul* benchmark [Seoul National University Real-Time Research Group 2008] (NonOpt), an optimized version using a cumulative temporal variable (Opt1), and a more highly optimized version changing the i, j, k nesting to i, k, j (Opt2). In all cases, matrices are stored in row-major order. For each case, Table I shows the locality type (self-temporal T, self-spatial S, or group G including both temporal and spatial); the ACDC behavior (instruction addresses to include in AC and data lines dynamically cached in DC); and the number of accesses, misses, and write-backs. To simplify the mathematical notation, we assume $n \times n$ aligned arrays,

```
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    for (k=0;k<n;k++)
      A[i][j]=A[i][j]+B[i][k]*C[k][j];
     /* st A     ld A     ld B     ld C */
```

(a) NonOpt

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

(b) Opt1

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

(c) Opt2

Fig. 3. Matrix multiplication codes.

Table I. Data References in Matrix Multiplication (Figure 3)

NonOpt	T	S	G	AC	DC	Accesses	Misses	Write-Backs
ld A	✓	✓	–	✓	✓	n^3	n^2/b	n^2/b
st A	–	–	✓	–	✓	n^3	0	0
ld B	–	✓	–	✓	✓	n^3	n^3/b	0
ld C	–	–	–	–	–	n^3	n^3	0
Opt1	T	S	G	AC	DC	Accesses	Misses	Write-Backs
ld A	✓	✓	–	✓	✓	n^2	n^2/b	n^2/b
st A	–	–	✓	–	✓	n^2	0	0
ld B	–	✓	–	✓	✓	n^3	n^3/b	0
ld C	–	–	–	–	–	n^3	n^3	0
Savings (NonOpt − Opt1): $2n^3 - 2n^2$							0	0
Opt2	T	S	G	AC	DC	Accesses	Misses	Write-Backs
ld A	–	✓	–	✓	✓	n^3	n^3/b	n^3/b
st A	–	–	✓	–	✓	n^3	0	0
ld B	–	✓	–	✓	✓	n^2	n^2/b	0
ld C	–	✓	–	✓	✓	n^3	n^3/b	0
Savings (NonOpt − Opt2): $n^3 - n^2$							$n^3 - n^3/b$	$(-n^3 + n^2)/b$

with n being a multiple of the cache line size b. Further, we place the write-back cases in the instruction that replaces the dirty lines.

It can be seen that Opt1 reduces the number of memory accesses (hits) and Opt2 provides a reduction in the number of misses. Using our proposed architecture, for the NonOpt and Opt1 codes, the instructions *ld A* (spatial and temporal reuse) and *ld B* (spatial reuse) would be given DC replacement permission—that is, their PC would be stored in the AC. This would allow them to use the data cache, and since *st A* (group-spatial reuse) is always executed after *ld A*, it would always hit. In these two codes, the C matrix is accessed by columns, which translates into no locality for small caches, and therefore no replacement permission is given to *ld C*. For the Opt2 code, all accesses have stride 1, so all loads would have replacement permission and all accesses would benefit from the data cache. As can be seen, the required size of the ACDC is small: 2 AC entries and 2 DC lines for the NonOpt and Opt1 cases, and 3 AC entries and 3 DC lines for the Opt2 case. Further, since each instruction with replacement permission can only replace its associated DC line, there is no pollution.

All of these benefits can be obtained by carefully selecting the specific instructions able to replace the data cache. To identify the optimal selection, we have extended Lock-MS, a previously proposed method for analysis and minimization of WCET.

4. LOCK-MS EXTENSION

In this section, we demonstrate how the proposed ACDC can be analyzed. Although many WCET analysis methods could be adapted, we have extended Lock-MS because it was designed for lockable instruction caches and fits very well to ACDC.

The aim of Lock-MS is to identify a selection of instruction lines from memory such that when locked into the instruction cache, the schedulability of the whole system is maximized [Aparicio et al. 2010, 2011]. For this, Lock-MS considers the resulting WCET of each task, the effects of interferences between tasks, and the cost of preloading the selected lines into cache. This method is based on integer linear programming (ILP) [Rossi et al. 2006]. Specifically, all requirements are modeled as a set of linear constraints and then the resulting system is minimized. In a similar way, our proposed ACDC requires a selection of instructions with permission to replace an associated line in the data cache. Thus, this extension allows us to analyze the worst-case behavior of any program when using our proposed ACDC. Note also that the analysis time of Lock-MS scales very well for large programs [Aparicio et al. 2011].

Previous work on Lock-MS has grouped all costs of a given instruction memory line in costs on instruction cache hit and miss. Considering the detailed costs in the case of our ACDC, we use the following organization for the instruction memory line k of path j of task i:

$$lineCost_{i,j,k} = fetch_{i,j,k} + exec_{i,j,k} + memory_{i,j,k}. \quad (1)$$

In this way, the fetch cost (*fetch*) includes exclusively the cost of retrieving the instruction, the memory access cost (*memory*) is the possible stalling of memory operations, and the execution cost (*exec*) is the remaining cost of the instruction. With this organization, the fetch and execution costs of a given memory line would be a literal translation of those in previous papers on Lock-MS [Aparicio et al. 2010, 2011]. Next we describe the constraints related to the data memory cost for the ACDC.

4.1. Constraints for the Hardware Structures

A memory instruction is modeled by the identifiers *pc* and *ref*. *pc* is the address of the instruction and *ref* represents its access(es) to a given data structure recognizable at compile time. In general, the memory reference does not appear in the code (source or compiled) as a constant address but rather as a series of operations to obtain the address. For instance, an access to an array inside a loop may have a memory reference based on the array base address, the size of elements, and the array index (based on the loop iteration). The association $\langle pc, ref \rangle$ cannot change—that is, a given memory instruction *pc* always accesses memory using a given reference *ref*. However, several memory instructions reusing the same data structure may have the same reference identifier. The theoretical basis used by compilers to match a data address expression with a reference and thus determine whether it is new or a data structure is being reused is outlined in Section 4.3.

We use binary variables to set whether an instruction has data cache replacement permission ($DRP_{pc,ref} = 1$) or not ($DRP_{pc,ref} = 0$). For a task i, the number of instructions with replacement permission must fit in the AC, and the number of data references to cache must fit in the DC:

$$\sum_{pc=1}^{MemIns} DRP_{pc,ref} = \; nInsDRP_i \; \leq AClines$$

$$\sum_{ref=1}^{MemRefs} DRP_{pc,ref} = \; nRefsDRP_i \; \leq DClines.$$

As well, the additional (ACDC) costs for each context switch must be considered. The costs regarding the AC would be those required to load the preselected PCs (those with data cache replacement permission) into the AC. The costs regarding the DC would be those needed to save the DC of a task being preempted and restore it when this task resumes its execution. Saving the DC means storing its tags and writing back the dirty contents, and restoring the DC means restoring the saved tags and refilling their corresponding data content. We consider the save and restore cost in all reuse cases:

$$ACDCswitchCost_i = ACpreloadCost \cdot nInsDRP_i + DCsave\_restCost \cdot nRefsDRP_i.$$

This ACDC context switch cost ($ACDCswitchCost_i$) times the maximum possible number of context switches in the selected scheduling policy is added to the WCET.

4.2. Memory Timing Constraints

The detailed data access cost can be described as the sum of the cost of each possible situation multiplied by the number of times that it occurs. The situations considered are data cache hits (*DChit*); data cache misses (*DCmiss*), with or without replacement; and line write-backs (*DCWB*). Since our proposed method of data cache management is instruction driven, we identify the accessed data by the instruction lines accessing these data (i.e., line k of path j of task i), and the resulting data memory cost is added to the line cost constraint (Equation (1)). A single memory line can, however, contain several memory instructions, and in such cases, data access costs must be considered separately. We use *nIns* to account for the number of instructions in a memory line, as in previous Lock-MS studies [Aparicio et al. 2011]:

$$\begin{aligned} memory_{i,j,k} = & \sum_{m=1}^{nIns_{i,j,k}} (DChitCost \cdot nDChit_{i,j,k,m} \\ & + DCmissCost \cdot nDCmiss_{i,j,k,m} + DCWBCost \cdot nDCWB_{i,j,k,m}). \end{aligned}$$

Considering fixed costs for the distinct possible cases of a data access, the only variables to define are those accounting for the number of such occurrences for a particular instruction m in a given memory line. Moreover, such occurrences are closely related. The number of hits is always the number of accesses (*nfetch*) to the instruction memory line (i.e., the number of times that the load or store instruction is executed) minus the number of data cache misses:

$$nDChit_{i,j,k,m} = nfetch_{i,j,k} - nDCmiss_{i,j,k,m}.$$

Further, the number of line write-backs depends on whether all of the instructions using the accessed data are loads or not. If all of them are loads, the number of write-backs is clearly 0, as the data are never modified. If, on the other hand, there is at least one store instruction using the accessed data, there will be write-backs. Since write-backs are performed on replacements, there will be as many write-backs as times the data line is replaced—in other words, the number of write-backs is equivalent to the number of data cache misses generated by those instructions with data replacement permission (e.g., see Table I):

$$\begin{aligned} nDCWB_{i,j,k,m} &= 0 \text{ if all instructions performing } ref \text{ are loads} \\ nDCWB_{i,j,k,m} &= nDCmiss_{i,j,k,m} \cdot DRP_{pc(i,j,k,m),ref} \text{ if at least one instruction} \\ &\quad \text{performing } ref \text{ is a store.} \end{aligned}$$

For clarity, this constraint shows a multiplication of variables, but it can be easily rewritten as a linear constraint by combining and simplifying it with the following constraint.

Considering any data cache, the number of misses of a given memory access depends on whether it has been cached before and has not been replaced. With our proposed ACDC, only instructions with data replacement permission can cache and replace, whereas those without DRP will always (*nfetch*) miss. Hence, the resulting constraints are

$$nDCmiss_{i,j,k,m} = ACDCmisses_{i,j,k,m} \cdot DRP_{pc,ref} + nfetch_{i,j,k} \cdot (1 - DRP_{pc,ref}),$$

where the $DRP_{pc,ref}$ will refer to the current instruction ($pc(i, j, k, m)$) in cases of self-temporal or self-spatial reuse and will refer to the previous reference in the case of group reuse. The only remaining value is the constant describing the number of misses in ACDC assuming data replacement permission, $ACDCmisses_{i,j,k,m}$.

4.3. Determine the Number of ACDC Misses

To determine the number of ACDC misses, we distinguish between references to scalar variables and to nonscalar variables. The number of misses for a given scalar variable depends exclusively on whether its address changes during program execution. For a global variable, the total number of misses is 1, corresponding to the first time it is referenced. For the remaining scalar variables, having a subroutine *sub* that is called $nStack_{sub}$ times using a stack frame address different from that of its previous call, the total number of misses is $nStack_{sub}$.

To calculate the number of misses of nonscalar variables, we consider loop nest data reuse and locality theory, briefly introduced later [Wolf and Lam 1991]. Each iteration in the loop nest corresponds to a node in the *iteration space*. In a loop nest of depth n, this node is identified by its iteration 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 loops. Let d be the 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 $Z^n \rightarrow Z^d$, 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 iteration variables corresponding to the kth array index. We consider this type of reference only. Next, the different types of reuse considered and their equations are described.

Self-Temporal Reuse (STR). It happens when a reference $A[H\vec{i}+\vec{c}]$ accesses the same data element in iteration $\vec{i_1}$ and $\vec{i_2}$—that is, $A[H\vec{i}_1 + \vec{c}] = A[H\vec{i}_2 + \vec{c}]$. The solution of the equation is the self-temporal reuse vector space $\ker(H)$. If it shows a vector $\vec{e}_i$ with all elements equal to 0 except one equal to 1 in position i, it means that there is temporal reuse in i (i.e., the iteration variable of loop i does not appear in any index function). In our case, a reference cannot have STR only, as it would mean that it is a scalar variable.

Self-Spatial Reuse (SSR). Let H_S be H with all elements of its last row replaced by 0—that is, a truncated H discarding the information about its last index: $A[in_1, in_2, \ldots, in_{d-1}]$. The self-spatial reuse vector space is then $\ker(H_S)$. If one of the solutions of this operator is a vector $\vec{e}_n$ with all elements equal to 0 except one equal to 1 in position n, with n being the last dimension of the iteration space, it means that there is spatial reuse. In other words, this vector indicates that the iteration variable of loop n does not appear in any other index function, so there will be accesses in sequence in the last dimension of A. In this article, we do not consider accesses with strides different from 1 or -1 element, so other strides would be accounted as misses.

Group-Temporal Reuse, Group-Spatial Reuse (GSR). In our particular case, two distinct references $A[H\vec{i} + \vec{c}_1] = A[H\vec{i} + \vec{c}_2]$ have both group-temporal and group-spatial reuse if and only if $\vec{c}_1 = \vec{c}_2$—that is, if both memory references are identical. One such reference will be SSR as well, so we classify it as SSR.

Table II. Reuse Matrices of Matrix Multiplication (Figure 3, Table I)

NonOpt	It. Sp.	H	STR	SSR	Ac.	Miss
ld A(i,j)	(i,j,k)	$\begin{pmatrix}1&0&0\\0&1&0\end{pmatrix}$	$(0\ 0\ 1)$	$(0\ 1\ 0)$ $(0\ 0\ 1)$	n^3	$\frac{n^2}{b}$
st A(i,j)	(i,j,k)	$\begin{pmatrix}1&0&0\\0&1&0\end{pmatrix}$	GSR	GSR	n^3	0
ld B(i,k)	(i,j,k)	$\begin{pmatrix}1&0&0\\0&0&1\end{pmatrix}$	$(0\ 1\ 0)$	$(0\ 1\ 0)$ $(0\ 0\ 1)$	n^3	$\frac{n^3}{b}$
ld C(k,j)	(i,j,k)	$\begin{pmatrix}0&0&1\\0&1&0\end{pmatrix}$	$(1\ 0\ 0)$	$(1\ 0\ 0)$ $(0\ 1\ 0)$	n^3	n^3
Opt1	It. Sp.	H	STR	SSR	Ac.	Miss
ld A(i,j)	(i,j)	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$	$\emptyset$	$(0\ 1)$	n^2	$\frac{n^2}{b}$
st A(i,j)	(i,j)	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$	GSR	GSR	n^2	0
Opt2	It. Sp.	H	STR	SSR	Ac.	Miss
ld A(i,j)	(i,k,j)	$\begin{pmatrix}1&0&0\\0&0&1\end{pmatrix}$	$(0\ 1\ 0)$	$(0\ 1\ 0)$ $(0\ 0\ 1)$	n^3	$\frac{n^3}{b}$
st A(i,j)	(i,k,j)	$\begin{pmatrix}1&0&0\\0&0&1\end{pmatrix}$	GSR	GSR	n^3	0
ld B(i,k)	(i,k)	$\begin{pmatrix}1&0\\0&1\end{pmatrix}$	$\emptyset$	$(0\ 1)$	n^2	$\frac{n^2}{b}$
ld C(k,j)	(i,k,j)	$\begin{pmatrix}0&1&0\\0&0&1\end{pmatrix}$	$(1\ 0\ 0)$	$(1\ 0\ 0)$ $(0\ 0\ 1)$	n^3	$\frac{n^3}{b}$

Table II shows the different references of the matrices for the three versions of matrix multiplication. It shows the iteration space, the H matrix, the self-temporal reuse vector space $\ker(H)$, and the self-spatial reuse vector space $\ker(H_S)$ for each reference. The number of accesses is obtained by multiplying the dimension of each index in the iteration space based on the lower and upper bounds of iteration variables in the loops. The last column shows the resulting DC misses *ACDCmisses*. For nonscalar references classified as GSR, the accessed addresses are identical to those of a previous reference (classified as SSR), and therefore they will already be cached, so $ACDCmisses = 0$. Algorithm 1 shows how to obtain the number of misses for nonscalar references classified as STR or SSR by exploring their reuse type on the different nesting levels of loops. Note that results in Table II are consistent with those in Table I, which were derived intuitively.

4.4. Structure-Based ILP

Previous constraints correspond to a path-based ILP model, as they specify all variables with subindexes for the path j in the task i. We use such notation to make them easier to understand. In general, however, the path information is not relevant, because data reuse is found in a given path most of the time. Hence, as long as the memory instructions are executed in the modeled sequence, previous constraints can be used in a structure-based ILP model by simply removing the path information (subindex j). The structure-based ILP model is much more compact and easy to resolve [Aparicio et al. 2011].

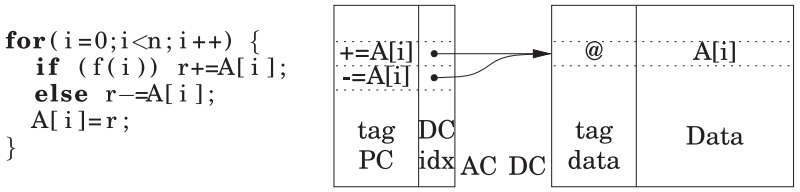


Fig. 4. Simple example of *cooperative* reuse involving different paths.

ALGORITHM 1: Algorithm to get *ACDCmisses* for STR/SSR nonscalar references

Input: H, *memLines(A)* # transformation matrix, data structure size
Output: *ACDCmisses*
1: **if** $e_n \in Ker(H_s)$ **then** # if *ref* has SSR (may have STR)
2: $i \leftarrow 0$
3: $H_i \leftarrow H$
4: **while** $e_{n-i} \in Ker(H_i)$ **do** # while ∃ STR
5: $i \leftarrow i + 1$
6: $H_i \leftarrow trunc(H_{i-1})$ # truncate column
7: **end while** # SSR in loop of depth i
8: $nmiss \leftarrow memLines(A)$
9: **for all** $e_k \in Ker(H_i)$ **do** # ∀ outer loops repeating accesses
10: $nmiss \leftarrow nmiss \times loopIter(k)$
11: **end for**
12: **return** *nmiss*
13: **else** # ref without reuse: always miss
14: **return** always
15: **end if**
Function *memLines*() returns the number of memory lines occupied by a given structure.
Function *loopIter*() returns the number of iterations of a given loop depth.

To detect and optimize the most basic data reuse cases involving *different* paths, we can generalize the previous constraints. Such basic cases involving different paths are those locality situations that may be found through different paths where the locality effects are the same independently of the followed path. In other words, all alternative paths have equivalent memory instructions accessing the same references. This means that different instructions may have data replacement permission on the same DC line. However, since these instructions are equivalent, they work *cooperatively* and do not pollute each other's cached data. Since the cache accesses are the same regardless of the path, previous constraints are valid, and they can be used in a structure-based ILP model. Figure 4 shows a simple example (references to array `A`) of this situation. In addition, the benchmark *integral* in the experiments that follow presents this behavior.

Finally, note that the proposed indexed ACDC structure allows an independent number of entries for the AC and the DC. This means that, anticipating the existence of such cooperative paths in tasks, the AC design could be larger than the DC. Since the AC latency is hidden by the main memory latency, the size of the AC can be increased, even if it implies more latency. As well, although for simplicity we assume a fully associative AC, its implementation is open to other configurations (direct mapped/set associative) with minor changes in the constraints for the AC hardware structure.

5. EXPERIMENTATION ENVIRONMENT

All of our experiments consist of modeling each task and system as linear constraints, optimizing the model and simulating the results in a rate-monotonic scheduler. Linear

Table III. Task Sets "Small" and "Medium"

	Task	Dir-Mem WCET (cy)	Period (Cycles)	Data Size (B)	Data Access (%)	Temporal (%) Self	Temporal (%) Group	Spatial (%) Self	Spatial (%) Group	Data Cached (%)	DC Lines
small	jfdctint	18,808	40,432	348	17.6	33.8	0	18.5	40.7	93.06	11
	crc	213,221	478,560	1,157	13.6	41.0	27.4	15.9	9.0	93.29	6
	matmul	834,359	2,169,448	4,828	29.8	0	0	51.2	23.8	74.94	5
	integral	1,587,329	6,534,486	168	42.1	13.3	86.6	0	0	99.91	1
medium	minver	16,793	43,902	4,752	33.0	9.3	25.4	28.7	5.4	68.88	16
	qurt	21,644	61,908	180	38.2	13.8	74.1	0.5	1.1	89.54	10
	jfdctint	18,808	62,066	348	17.6	33.8	0	18.5	40.7	93.06	11
	fft	4,742,498	14,792,660	528	22.9	11.4	75.7	5.7	0	92.82	12

constraints follow the Lock-MS model to minimize the WCET [Aparicio et al. 2011]. The feasibility of such a system can be tested in a number of ways [Sha et al. 2004]. *Response time* analysis is one of these mathematical approaches, and it is used as our main multitask metrics. This approach is based on the following equation for independent tasks:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j, \tag{2}$$

where R_i is the response time, C_i is the WCET plus the overheads due to context switches in the worst case, and T_i is the period of each task i. It is assumed that tasks are ordered by priority (the lower the i, the higher the priority). This equation provides the response time of each task after a few iterations. A task meets its real-time constraints if $R_i \leq T_i$.

Table III lists the two sets of tasks used in our experiments. Benchmarks include JPEG integer implementation of the forward DCT, CRC, matrix multiplication, integral computation by intervals, matrix inversion, computation of roots of quadratic equations, and FFT from the *SNU-RT Benchmark Suite for Worst Case Timing Analysis* [Seoul National University Real-Time Research Group 2008]. The "small" and "medium" task sets have been used in previous studies with similar periods [Puaut and Decotigny 2002; Aparicio et al. 2011]. Sources have been compiled with GCC 2.95.2 -O2 without relocating the text segment (i.e., the starting address of the code of each task maps to cache set 0). Our ILP models the resulting binary code. The option -O2 implies *fomit-frame-pointer* (the frame pointer is not used). In addition, the stack can grow only once in each routine (*Stack Move Once* policy).

The "Dir-Mem WCET" in Table III refers to a system without caches or buffers, having direct access to separate embedded SRAMs (eSRAMs) for instructions and data, and it has been computed without context switches. For this cacheless system, task periods have been set so that the CPU utilization is 1.5 for the small task set and 1.35 for the medium task set. These values have been slightly increased with respect to the original source by Puaut and Decotigny [2002] to constrain the schedulability of a/our more powerful memory hierarchy (ACDC+IC+LB+Prefetch). Data size, in bytes, includes the static data size and the maximum size reached by the stack. The remaining columns in this table show different percentages of data access and locality, as well as the data cached and the required DC lines when using our proposed ACDC. These values are discussed in Section 6.2.

The target instruction set architecture considered in our experiments is ARMv7 with instructions of 4 bytes. We use separate instruction and data paths, each one using its own 128 KB eSRAM as main memory. The cache line size is 16 bytes (4 instructions). The instruction cache (IC) size is varied from 64 bytes (4 sets) to 1KB (64 sets), all

Table IV. Timing (Cycles) Considered in Different Operations (Figure 1(b))

Fetch hit	1 (IC/LB/PB access)
Fetch miss	1+6 (IC/LB/PB + memory access)
Fetch by prefetch	1 to 6
Data access without ACDC	1+6 (addr. computation + memory)
① ACDC hit	1+1 (addr. + ACDC access)
② ACDC miss without replacement	1+1+6 (addr. + ACDC + memory)
③ ACDC miss and replacement without write-back	1+1+6 (addr. + ACDC + memory)
④ ACDC miss and replacement with write-back	1+1+6+6 (ACDC miss + memory)

direct mapped. All tested memory configurations include an instruction line buffer (LB) keeping the most recently fetched line, and some of them include a form of sequential-tagged instruction prefetch that keeps a single prefetched line in a prefetch buffer (PB) as in previous studies [Puaut and Decotigny 2002; Aparicio et al. 2010, 2011]. It is important to note that a careful modeling of the instruction supply is essential to study the real impact of the ACDC on the system, because instruction fetch delays and interactions with data references set the worst path and its WCET. Our proposed ACDC structure is composed of a 16-way fully associative data cache (DC) with 16 bytes per line and a 16-way fully associative address cache (AC) with 4 bits per entry, plus their required tags (Figure 1(a)). To compute memory circuit delays, we have used Cacti v6.0 [Muralimanohar et al. 2007], a memory circuit modeling tool, assuming an embedded processor built in 32nm technology and running at a processor cycle equivalent to 36 FO4.[1] All tested caches meet the cycle time constraint. Further, the access time of each eSRAM is 6 cycles if we choose to implement it with low standby power transistors. The specific cost of the instruction fetch and data access (associated with the specific behaviors detailed in Figure 1(b)) can be seen in Table IV. Note that on data misses, our proposed data cache performs worse than a system without data cache. Moreover, a data hit is only four times better than a data miss. In other words, we assume a base system with a very good performance to truly test our ACDC. Other studies consider off-chip main memories and assume a higher miss penalty, such as 38 cycles [Vera et al. 2003] or 64 cycles [Gonzalez-Alberquilla et al. 2010]. Clearly, in these systems, the room for improvement is much higher. Thus, our results may be seen as a lower bound on performance, which would rise if the T_{miss}/T_{hit} ratio were increased.

6. REAL-TIME RESULTS

In this section, we combine our proposed data cache structure (Figure 1(a)) with the instruction fetch components (lockable instruction cache, line buffer, and instruction prefetch) [Aparicio et al. 2010, 2011]. Figure 5 shows the response time (Equation (2)) speed-up of the lowest-priority task relative to a baseline system with an instruction LB but no caches (first bar of group 0). As a reference, this baseline is schedulable, and its response times are 0.85 and 0.60 times the largest period for the small and medium task sets. The tested memory configurations combine the previously introduced components and policies, namely LB, PB, IC, and our proposed data cache (ACDC). The IC is a dynamically locked instruction cache with contents selected using the Lock-MS method [Aparicio et al. 2010, 2011]. The first bar group (labeled 0) assumes no IC, and the remaining four bar groups vary the IC size from 64 to 1,024 bytes. The ACDC size (256 bytes) is not varied, because even with such small size, tasks use only a subset of its lines. The first bar (LB+DC Lock-MU) represents a system with an instruction

[1]A fan-out-of-4 (FO4) represents the delay of an inverter driving four copies of itself. A processor cycle of 36 FO4 in 32nm technology would result in a clock frequency of around 2.4GHz, which is in line with the market trends [Microprocessor-Report 2008].

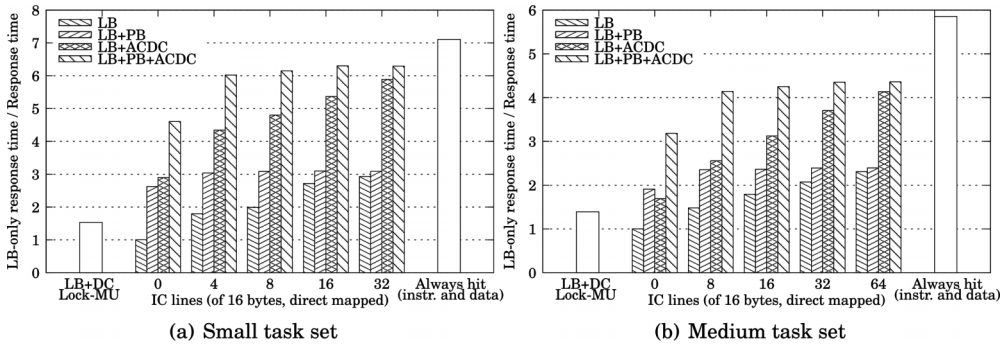


Fig. 5. Response time speed-up of the lowest priority task for several memory configurations (normalized to a cacheless LB-only system).

LB and a statically locked data cache of the same size (256 bytes) using the Lock-MU method on data [Puaut and Decotigny 2002]. Such a configuration exploits temporal locality but not spatial locality, whereas ACDC exploits both locality types. Finally, bar *Always hit* represents an ideal system always hitting in both instructions and data. Response times of this ideal system are 0.12 and 0.10 times the largest period for the small and medium task sets. This is an unreachable bound but provides an easily reproducible reference. Performance of the considered systems is discussed later.

6.1. Instruction Cache and Prefetch

Regardless of task set size, systems with instruction prefetch (LB+PB, LB+PB+ACDC) are relatively insensitive to IC size, whereas those without prefetch (LB, LB+ACDC) depend much more on it. As expected, the benefit of prefetching decreases as the IC size grows. Thus, instruction prefetch is especially interesting to improve systems with small instruction cache sizes. In addition, it improves tasks with large sequences of instructions executed a few times. Adding our ACDC improves any memory configuration. Independent of the IC size, the speed-up roughly doubles (LB vs. LB+ACDC and LB+PB vs. LB+PB+ACDC) for the small task set, whereas for the medium task set the speed-up is roughly 1.5.

6.2. ACDC Analysis

As outlined earlier, across all considered memory configurations and task sets, ACDC enhances worst-case performance by a factor of 1.5 to 2. To obtain further insight into how individual tasks benefit from spatial and temporal ACDC reuse, we can consider Table III again. Note that the percentage of data that has permission to be placed in the DC ("Data Cached (%)" column) is high. The average is 90.30% for the small task set and 86.08% for the medium. For the small task set, the only task for which not particularly high efficiency is achieved (74.94%) is the matrix multiplication. It accesses a matrix by columns (stride > 1), which prevents any small data cache from being effectively exploited. The *matmul* benchmark specifies an i, j, k loop nesting without a temporal variable outside the deepest loop for calculating the row times column multiplication. As described earlier, a better implementation would set an i, k, j loop nesting with a temporal variable outside the deepest loop. Implementing such an optimization, our ACDC would cache much more accesses (99.01%), as all matrix references would have stride 1. Similarly, the medium task set has also a matrix benchmark (matrix inversion) with stride > 1. Despite these two tasks, our results reach 88% and 75% of the always-hit case for the small and medium task sets having a DC of just 256 bytes.

The specific number of data cache lines used by each benchmark can be seen in the last column of Table III ("DC Lines"). For instance, 99.91% of data is captured with a single DC line (128 bits) for the *integral* benchmark and 93.29% with six DC lines (96 bytes) for the *crc* benchmark. Note that this number is very small, especially when considering that it is independent of the size of the data structures in the task. In general, each data structure should require a single DC line. Additionally, remember that DC lines are shared between tasks, and therefore an optimal sizing would only consider the more demanding task (e.g., 11 lines for the small task set). Furthermore, although we assume (and apply) a context switch penalty on each and every context switch, note that a clever distribution of the DC lines used by each task could avoid (several of) such penalties, using the ACDC as if it were partitioned. Thus, our proposed data cache is very suitable for embedded systems, where size may matter.

Additionally, the sixth column of Table III ("Data Access (%)) shows the percentage of data accesses over the total memory accesses (data+fetch) for each task. It can be seen that between 13% and 42% of memory accesses are data accesses, which represents an important factor in the WCET calculation. The seventh and eighth columns ("Temporal" and "Spatial") show the percentage of data accesses managed by ACDC, divided into self- and group locality. As can be seen, some tasks have temporal locality or spatial locality only, but most of them have both types. This means that efficiency when using a unified data cache is higher than other structures intended to manage them separately.

Our results use a timing with a T_{miss}/T_{hit} ratio of 4. Experiments would not be schedulable using a ratio of 38, which prevents direct comparisons with other methods [Vera et al. 2003]. On the other hand, processor utilization values can be compared. The static locking system (LB+DC Lock-MU) provides utilization values of 1.74 and 1.25 for the small and medium task set, whereas the utilization for LB+ACDC system without IC (0.86 and 0.72) is similar to that found where tasks can lock/unlock the data cache dynamically [Vera et al. 2003]. However, our results are achieved with a much smaller data cache, without modifying the code of tasks and without partitioning. Moreover, in the case of tasks using many data structures (and requiring more than our current 16 DC lines), we could easily follow a similar dynamic locking behavior—that is, specifying different data replacement permissions (AC contents) for different task phases. Such an approach is explored in Section 7.3.

6.3. Partitioning

In this section, we analyze how much improvement would come from partitioning resources (i.e., without costs on context switches). To focus on the effects of intertask interferences, instead of partitioning a predefined size for the instruction and data caches, we replicate all buffers and caches for each task. In this way, we avoid the problems associated with selecting adequate partitions and just provide an upper performance bound. In other words, any partitioning technique will perform as well as or less well than a per-task hardware replication.

Figure 6 shows the same memory configurations as earlier. In this case, the vertical axis shows the response time speed-up due to full replication (shared/replicated). In other words, values of the response time with shared hardware include costs of saving/restoring the ACDC at context switches, whereas values with full replicated hardware do not. Note that without such penalties, the resulting replicated system can use the proposed hardware much more effectively—that is, lines that were not cached because their associated penalties on context switches were too large can be cached now. Nevertheless, Figure 6 shows very marginal benefits when replicating hardware. Both for the small and medium task sets, results show improvements of less than or around 10%. Improvements are smaller ($<$2% and $<$8% for the small and medium task sets, respectively) when prefetch is enabled. Differences between the small and

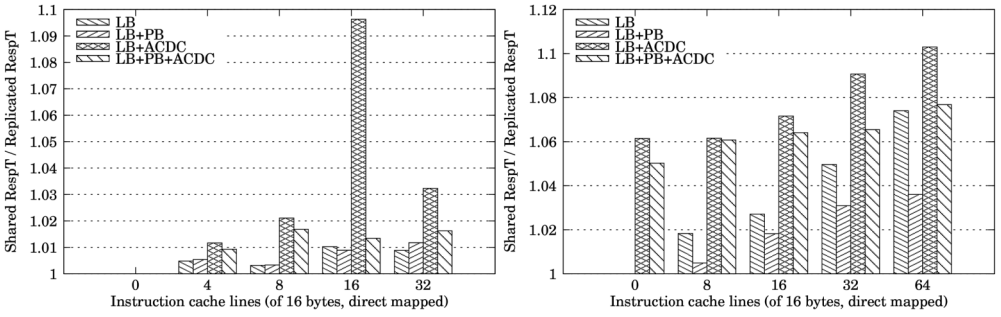


Fig. 6. Response time speed-up of full resource per-task replication of several memory configurations.

Table V. MiBench Programs Used in Our Experiments

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg (enc)	ispell	dijkstra	blowfish (enc)	adpcm (enc)
bitcount	jpeg (dec)	rsynth	patricia	blowfish (dec)	adpcm (dec)
quicksort	mad	stringsearch	(blowfish)	rijndael (enc)	fft
susan (corners)	tiff2bw		(sha)	rijndael (dec)	fft-inv
susan (edges)	tiffdither			sha	
susan (smooth)	tiffmedian				

medium task sets can be attributed to the fact that the medium task set involves much more context switches, so the associated penalties are also higher.

As outlined earlier, replication offers the highest performance bound on partitioning, because partitioning techniques without increasing the ACDC size would appear to have a much smaller structure for each task, which would result in a lower hit ratio. Thus, taking cost into account, it seems that such marginal benefits are not worth replication or partitioning.

7. USING ACDC FOR LARGE EMBEDDED PROGRAMS

Previous experiments and results show the worst-case performance of the ACDC with real-time benchmarks. Such results show that the small ACDC size is enough for real-time kernels. In this section, we analyze the data reuse of several embedded programs to foresee how they would perform if translated into real-time tasks.

We use the programs in Table V (*MiBench* embedded benchmark suite [Guthaus et al. 2001]) for this analysis. This suite is composed of a set of commercially representative programs. They include unbounded loops, which means that they cannot be analyzed for worst-case performance, as in our previous sections. As pointed out by the authors of MiBench, embedded applications generally have small data memory segments, but programs in MiBench present large variations in their data segment due to the large number of immediate values (constants) embedded in source code. These large data segments should stress the ACDC considerably. Moreover, MiBench programs sometimes have large data tables for lookups (i.e., arrays accessed by a data-dependent index) or interpolation, which are inherently nonpredictable. Furthermore, some of these programs contain manual optimizations in their source code, which may be inconvenient for specific targets, as they obscure the intended semantics. For instance, loop unrolling may harm performance in processors with a *zero-overhead loop buffer* [Uh et al. 1999] and may demand more ACDC lines than those required using a loop construct. Although some programs include preprocessor directives to adapt the

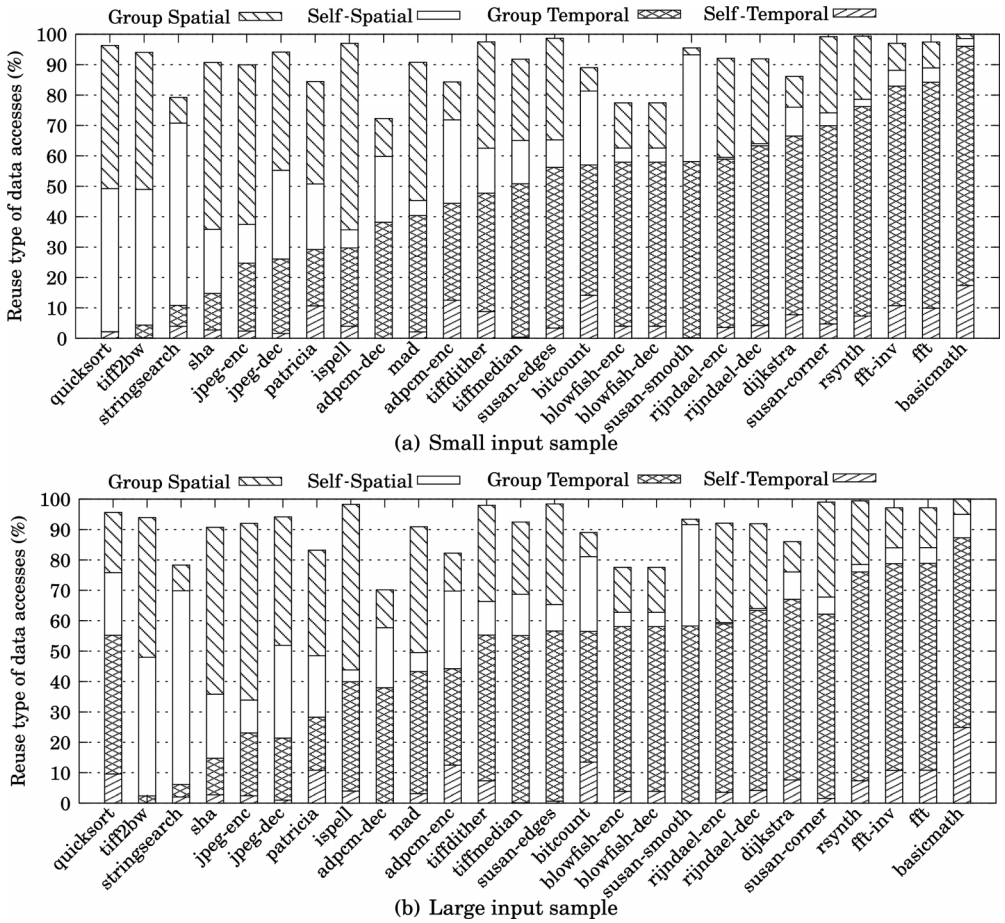


Fig. 7. Reuse types found in the data accesses of MiBench programs.

generated executable, we use the statically linked ARM binaries provided by MiBench. All of these programs include two input samples (small and large) and specific options for their execution. Our experiments use those samples and options. Finally, all data memory accesses have been considered in our experiments, except those in input/output functions.

7.1. Reuse Characterization

Figure 7 shows the percentage of the different reuse types in the data accesses of the analyzed programs, considering that memory is organized in lines of 16 bytes. Each bar, representing a particular program, is divided (from bottom to top) into self-temporal, group-temporal, self-spatial, and group-spatial reuse. Essentially, self-temporal reuse represents accesses to scalar variables by a single load/store instruction. Group-temporal reuse represents accesses to scalar variables by multiple load/store instructions—that is, one instruction caches a scalar variable, and other instructions find it already cached. Self-spatial reuse essentially shows array walks performed by a single instruction. For instance, an array walk caches a memory line containing n elements and hits in $n - 1$ elements in this memory line. Finally, group-spatial reuse represents array walks with several instructions involved—that is, an instruction

caches a memory line containing n elements, and several instructions access these elements. The remaining distance until 100% contains those accesses not having spatial or temporal reuse, as described earlier. Bars are ordered by increasing temporal reuse in Figure 7(a) and follow the same order in Figure 7(b) for an easier comparison.

First of all, note that the programs present a wide variability in reuse types, from programs with almost no temporal reuse to programs showing 96% of temporal reuse. In addition, note that there is significant variability between self- and group spatial reuse, whereas most temporal reuse is group reuse. This means that array walks can be used by a single instructions or by several instructions, depending on the algorithm, but scalar variables are commonly used by many instructions and not a single one. For instance, the source code of *basicmath* contains several scalar variables that are being reused continuously along the code and a small array of three elements (double floats, occupying two memory lines). This is translated into a major percentage of group-temporal reuse. On the other hand, spatial reuse depends much more on particular implementations. For instance, the algorithms implemented in *susan-corners* and *susan-edges* are very similar to that of *susan-smooth*. Whereas *susan-smooth* implements an array walk by means of a loop containing a memory access, the other two algorithms implement it by a long sequence of explicit memory accesses in the source code. Although both options produce similar array walks, such accesses are classified as self-spatial in *susan-smooth* and group spatial in *susan-corners* and *susan-edges*.

Figure 7 shows that programs present a very similar percentage of reuse types independently of the size of the input sample. Such percentages can be seen as a reuse fingerprint of the program, showing that data reuse depends mainly on the algorithms (instructions) and not on the data. Apparently, *quicksort* shows different percentages between the small and large input samples, but actually two different programs (*qsort_small* and *qsort_large*) are used in MiBench, as our results suggest. Thus, capturing the data reuse by means of the instructions, as our ACDC does, generally should provide very good results. Additionally, it confirms that results are independent of the size of the data structures in tasks, as they are managed by the same instructions—that is, they work with the same reuse patterns.

7.2. Impact of the ACDC Line Size

Although most benchmarks present a high percentage of data reuse, exploiting it (i.e., translating it into data hits) depends on the cache structure. In general, each parameter in a cache has its own trade-offs. For instance, designs of fully associative caches with more than 16 ways are possible, but they would require a careful adjust to the processor cycle time. Considering our ACDC proposal, enlarging the line size benefits spatial locality and has no significant drawbacks. Moreover, the worst-case analysis with a larger line size would be simpler and faster, as the granularity is coarser.

The reuse percentages considering memory lines of 16 bytes, as calculated previously, always increase when using larger sizes. Strictly speaking, reuse should be associated to memory elements (variables, elements in arrays, etc.) considering the specific size of each element. Such reuse analysis would be independent of the considered line size. However, it would not accurately reflect the actual reuse that caches can exploit, because caches work with memory blocks fitting in lines of a fixed size. The top marks in Figure 8 show the reuse bounds calculated as earlier (height of bars in Figure 7) but considering a memory organization in lines of 64 bytes. This provides an upper bound on data reuse—that is, the hit ratio that would be achieved using an ideal ACDC with unlimited DC lines of 64 bytes each.

To see the actual ACDC behavior, bars in Figure 8 show the hit ratio that can be achieved by our proposed ACDC (16 DC lines) with line sizes of 16, 32, and 64 bytes—that is, with a total ACDC size of 256, 512, and 1024 bytes. In this case, results depend

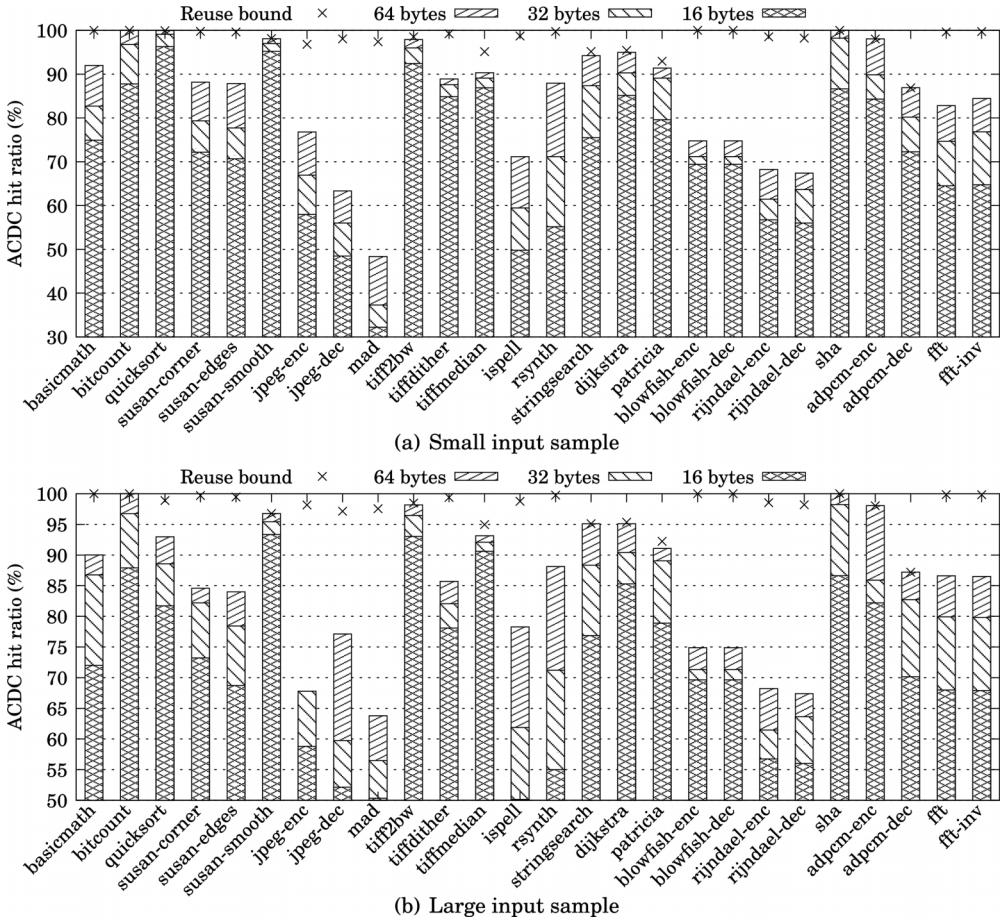


Fig. 8. ACDC hit ratio of MiBench programs for different ACDC line sizes (always with 16 DC lines).

much more on the programs. Whereas some of them, such as *bitcount*, *dijkstra*, or *sha*, are able to achieve a hit ratio very close to the reuse bound, others, such as *jpeg*, *mad,* and *rijndael*, are quite far. Moreover, it is hard to say in advance if a program will achieve a high hit ratio, because those with worse results do not seem to present any particular reuse fingerprint (Figure 7).

To provide further insight into how to reach the reuse bound, Table VI shows the number of DC lines that would be required to achieve a hit ratio equal or higher than 90% of the reuse bound. For each program, it shows the required number of DC lines considering sizes of 16, 32, and 64 bytes, both for the small and large input samples.

As outlined previously, increasing the line size improves results. This means that when the line size increases, less DC lines are required to achieve a similar hit ratio, as can be seen in Table VI. However, note that the reuse bound may also increase when enlarging this size (differences between the height of bars in Figure 7 and top marks in Figure 8). Hence, when the hit ratio is very close to the reuse bound (e.g., in *bitcount*, *adpcm*) or when the increment of the reuse bound is large (e.g., in *blowfish*), the number of DC lines to reach 90% of the reuse bound may also grow, as the amount of hits in absolute numbers is also increased.

Table VI. Number of DC Lines Required to Achieve a Hit Ratio Equal or Higher Than 90% of the Reuse Bound, for the Small and Large Input Samples, Using Different ACDC Line Sizes

	16B		32B		64B			16B		32B		64B	
Program	sml	lrg	sml	lrg	sml	lrg	Program	sml	lrg	sml	lrg	sml	lrg
basicmath	38	32	24	19	15	14	rsynth	49	48	28	28	16	16
bitcount	13	13	9	10	14	14	stringsearch	13	12	11	9	10	9
quicksort	14	20	13	17	12	14	dijkstra	7	7	6	5	6	5
susan-corner	39	33	28	27	18	24	patricia	13	13	10	10	10	11
susan-edges	40	40	27	33	16	28	blowfish-enc	16	16	45	45	49	49
susan-smooth	5	5	6	6	7	7	blowfish-dec	16	16	44	45	49	49
jpeg-enc	51	55	39	43	30	28	rijndael-enc	243	243	148	149	59	58
jpeg-dec	57	53	40	35	32	29	rijndael-dec	245	245	148	149	60	60
mad	115	73	83	55	64	44	sha	13	13	12	12	8	8
tiff2bw	4	4	4	4	5	4	adpcm-enc	7	6	7	7	11	11
tiffdither	23	35	18	25	18	19	adpcm-dec	4	4	5	5	10	12
tiffmedian	15	13	14	12	14	12	fft	43	38	31	29	21	18
ispell	68	84	52	52	35	32	fft-inv	44	42	28	29	21	18

One of the key points of the ACDC is that its performance is independent of the size of the data structures in tasks. Indeed, contrary to conventional caches, it is not rare to see the ACDC performing better when the size of such structures is large. This counterintuitive behavior can clearly be seen in *basicmath*, *jpeg-dec*, *mad*, *tiffmedian*, *stringsearch*, and *fft*, as well as in other programs to a lesser extent, where the large input sample requires fewer DC lines than the small input sample.

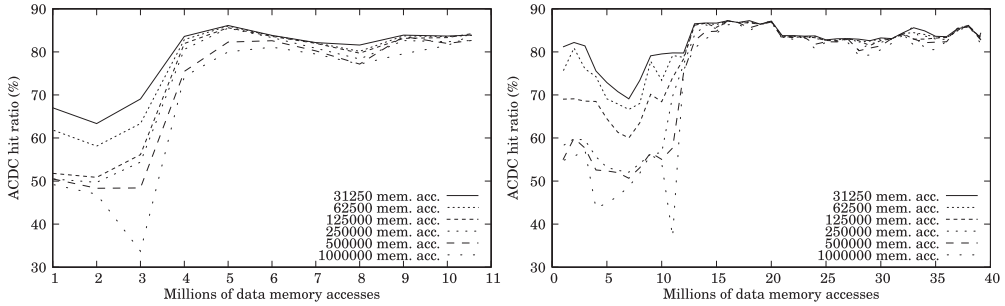
In general, values equal to or lower than 16 in Table VI mean that reuse can be effectively exploited with the (static) ACDC—that is, a selection of PCs stored in AC for the whole program runtime. On the other hand, larger values correspond to programs with a low hit ratio in Figure 8 and indicate that it would require an ACDC with more DC lines. However, programs with values larger than 16 should be studied further, as they probably present several phases in their execution and may require dynamic ACDC reconfigurations.

7.3. Dynamic ACDC Reconfigurations

Programs usually present different phases in their execution. Each phase may have its own data access patterns and may work with different data structures. Thus, in such cases, each program phase could use a specific ACDC configuration. Such an approach has been used previously on instruction caches [Vera et al. 2003; Puaut 2006].

MiBench suite contains complete programs, which include phases of processing input options, data initialization, data processing, output processing, and so forth. Hence, some of these programs could benefit from reconfiguring the ACDC during the program execution. This would increase the hit ratio and reduce the required ACDC lines in each phase.

To adjust the ACDC for program phases, the programmer/compiler should place a mark whenever the program changes its data access behavior. Such marks would trigger the ACDC reconfiguration—that is, replace the PCs currently stored in the AC (those with data cache replacement permission) by a new set of PCs more suitable for the next phase. In general, studying program phases is encouraged for any program with a low hit ratio. For instance, the hit ratio achieved by the *jpeg encoding* program is far from its corresponding reuse bound (Figure 8), so it is a good candidate for testing dynamic ACDC reconfigurations. To perform fair and reproducible experiments, we avoid marking the program manually. Instead, dynamic ACDC reconfigurations are applied at regular intervals of a specific number of memory accesses. Shorter intervals



(a) Small input sample (b) Large input sample

Fig. 9. ACDC hit ratio in the *jpeg-enc* program with dynamic ACDC reconfigurations at regular intervals.

imply more specific AC contents for each interval. Hence, if the program has phases, this should be translated into more hits in each interval.

Figure 9 shows the resulting ACDC hit ratio when using dynamic ACDC reconfigurations in the *jpeg encoding* program at regular intervals of data accesses. The ACDC considered has 16 DC lines of 16 bytes each, and the ACDC configuration of each interval has been obtained in a heuristic way. Decreasing the length of the intervals means more specific ACDC configurations. Hence, performance should improve even more with shorter intervals, converging to a point where the latency of all reconfigurations equals the benefits of using these specific ACDC configurations. The rightmost 6 million accesses in Figure 9(a) and the rightmost 25 million in Figure 9(b) are very similar independently of the ACDC reconfiguration intervals. This means that reducing the ACDC reconfiguration interval provides minor benefits. In general, in these situations, a static ACDC should achieve good results and reconfigurations would not be required. On the other hand, at the beginning of execution, results depend heavily on the ACDC reconfiguration interval. Without dynamic ACDC reconfiguration, these transient phases are ignored because they have a low impact when considering the whole program. However, they account for one third of memory accesses, and ignoring them has a significant impact on the average hit ratio. Hence, dynamic reconfiguration enables the ACDC to adapt to this initial changing behavior.

Figure 10 presents experiments similar to Figure 8 but shows the best ACDC hit ratio when including experiments with dynamic ACDC reconfigurations at regular intervals. The largest tested interval performs an ACDC reconfiguration at each 1 million data accesses. Additional intervals have been tested by halving this length over and over, as in Figure 9, but this time until reaching intervals of 3,906 data accesses. Nevertheless, to obtain the best results, ACDC reconfiguration triggers should be placed by the programmer/compiler at phase changes.

In approximately one third of the programs (those with good results in Figure 8), ACDC reconfigurations are not needed. On the other hand, 9 programs using the small input sample and 11 programs using the large input sample achieve their best hit ratio with a medium-grain interval (31,250 data accesses). Finally, *jpeg-enc*, *jpeg-dec*, *mad*, *tiffdither*, *ispell,* and *rsynth* (also *tiffmedian* with the large input sample) perform better with reconfigurations using the finest-grain tested intervals.

In most cases, the achieved hit ratio goes beyond 90% and many times reaches the reuse bound. There are three programs (*mad*, *blowfish*, and *rijndael*) whose achieved hit ratio remains between 65% and 80%. Both *blowfish* and *rijndael* use large (4KB) lookup tables, which account for around 30% of the data accesses in each program, and *mad* uses similar structures. References to such data structures are inherently nonpredictable, so they are hard to manage. This can be seen in Table VI, where

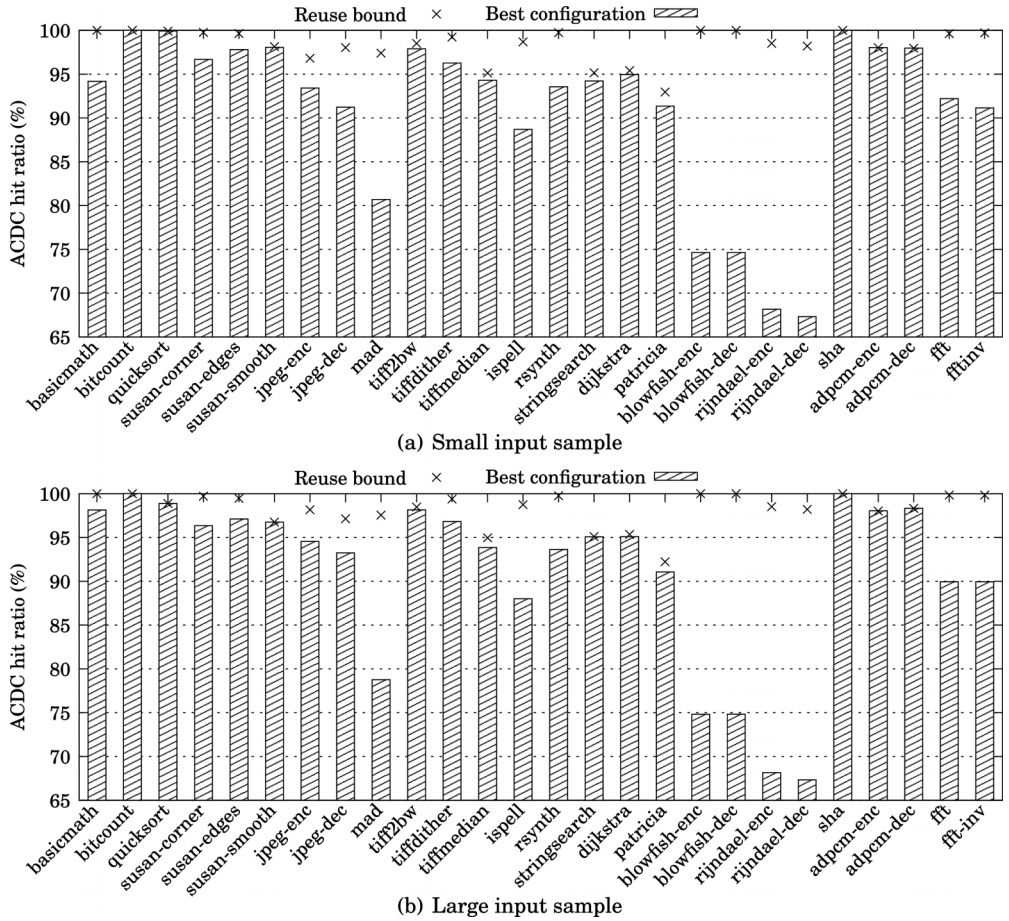


Fig. 10. Best ACDC hit ratio for each MiBench program, allowing dynamic ACDC reconfigurations.

these programs appear to require many more DC lines than the others. To achieve predictability and a higher hit ratio, they would require that the temporal locality to the whole data structures would be captured at the same time in a large cache structure, because they have no regular access pattern. Thus, possible approaches could be mapping such structures to a scratchpad, specific cache designs for this type of accesses, and so forth. Evaluation of such approaches will be addressed in future work.

8. CONCLUSIONS

In this article, we propose a small instruction-driven data cache (ACDC) that effectively exploits both the temporal and the spatial locality. It works by preloading at each task, switching a list of instructions with data cache replacement permission, which are assigned their own data cache line. Since each memory instruction replaces its own data cache line, pollution is avoided and performance is independent of the size of the data structures in tasks. In general, each data structure should require a single data cache line. Moreover, by its nature, it guarantees that data cache lines cannot be replicated. Worst-case performance with this ACDC is relatively easy to analyze (similar level of difficulty to a locked instruction cache) but maintains its dynamic behavior, because contents are replaced similarly to conventional caches. Further, since the selection of instructions with replacement permission is based on the reuse theory,

explicit sequences of data access addresses are not required and references to unknown addresses can be analyzed.

We have extended the WCET analysis/minimization method Lock-MS to include the ACDC. To study the real impact of the ACDC in the system, experiments include different combinations of instruction cache, instruction line buffering, and instruction prefetch, whose interaction with ACDC sets the worst path and its WCET. Our results show a high percentage of cached data on the studied benchmarks (around 93% in most cases), reaching 99.9% in the integral calculation benchmark and values of around 70% in inefficiently programmed matrix benchmarks. Such results are obtained with an ACDC of 256 bytes, although most tested benchmarks do not fully use such capacity. This small size makes our proposal especially interesting for embedded devices. In our experiments, worst-case performance with the ACDC is roughly double that without the ACDC, approaching (75% to 89%) the ideal case of always hitting in both instructions and data. These values are similar to those reported for methods where tasks can lock/unlock the data cache dynamically, but our results are achieved with a much smaller data cache, without modifying the codes of tasks and without partitioning.

We repeated our real-time experiments assuming that each task had its own replicated hardware to bound the benefits of using ACDC in a partitioned environment. Our results show only marginal improvements (of less than or around 10%), stating that the added cost of partitioning makes this option uninteresting if ACDC is implemented.

Finally, a reuse characterization has been performed on MiBench programs. Results show that such programs present a high reuse in general, which translates into a high ACDC hit ratio in most cases. We also show how to increase such hit ratio both by increasing the ACDC line size and by performing dynamic ACDC reconfigurations.

As future work, temporal reuse of nonscalar variables is still an unsolved problem. Such reuse includes regular patterns, such as accessing by columns a row-major ordered matrix, and accesses with nonregular patterns, such as lookup tables. Those with regular patterns present a high miss ratio, even with conventional data caches, and irregular patterns imply nonpredictable accesses.

REFERENCES

S. Altmeyer, C. Maiza, and J. Reineke. 2010. Resilience analysis: Tightening the CRPD bound for set-associative caches. *ACM SIGPLAN Notices* 45, 4, 153–162.

L. C. Aparicio, J. Segarra, C. Rodríguez, J. L. Villarroel, and V. Viñals. 2008. Avoiding the WCET overestimation on LRU instruction cache. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 393–398.

L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. 2010. Combining prefetch with instruction cache locking in multitasking real-time systems. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 319–328.

L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. 2011. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *Journal of Systems Architecture* 57, 695–706.

M. Geiger, S. McKee, and G. Tyson. 2005. Beyond basic region caching: Specializing cache structures for high performance and energy conservation. In *Proceedings of the International Conference on High-Performance and Embedded Architectures and Compilers*. 102–115.

S. Ghosh, M. Martonosi, and S. Malik. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems* 21, 4, 703–746.

A. González, C. Aliagas, and M. Valero. 1995. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the International Conference on Supercomputing*. 338–347.

R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, and F. Tirado. 2010. Stack filter: Reducing L1 data cache power consumption. *Journal of Systems Architecture* 56, 12, 685–695.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization*. 3–14.

H. S. Lee and G. S. Tyson. 2000. Region-based caching: An energy-delay efficient memory architecture for embedded processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 120–127.

Y. T. S. Li, S. Malik, and A. Wolfe. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the IEEE Real-Time Systems Symposium*. 254–264.

T. Lundqvist and P. Stenström. 1999. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems* 17, 2–3, 183–207.

A. Martí Campoy, Á. Perles Ivars, and J. V. Busquets Mataix. 2001. Static use of locking caches in multitask preemptive real-time systems. In *Proceedings of the IEEE Real-Time Embedded System Workshop*.

A. Martí Campoy, Á. Perles Ivars, F. Rodríguez, and J. V. Busquets Mataix. 2003a. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*.

A. Martí Campoy, S. Sáez, Á. Perles Ivars, and J. V. Busquets Mataix. 2003b. Performance comparison of locking caches under static and dynamic schedulers. In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*.

Microprocessor-Report. 2008. Chart watch: High-performance embedded processor cores. *Microprocessor Report* 22, 26–27.

N. Muralimanohar, T. Balasubramonian, and N. P. Jouppi. 2007. *Cacti 6.0: A Tool to Understand Large Caches*. Technical Report. University of Utah and Hewlett Packard Laboratories.

I. Puaut. 2006. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 217–226.

I. Puaut and D. Decotigny. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. 114.

I. Puaut and C. Pais. 2007. Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. 1–6.

R. Reddy and P. Petrov. 2007. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 198–207.

F. Rossi, P. V. Beek, and T. Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.

Seoul National University Real-Time Research Group. 2008. SNU-RT benchmark suite for worst case timing analysis.

L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. 2004. Real time scheduling theory: A historical perspective. *Real-Time Systems* 28, 101–155.

V. Suhendra and T. Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th Design Automation Conference*. 300–303.

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

G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. 1995. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*. IEEE, Los Alamitos, CA, 93–103.

G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, and V. Cao. 1999. Effective exploitation of a zero overhead loop buffer. *ACM SIGPLAN Notices* 34, 7, 10–19.

X. Vera, B. Lisper, and J. Xue. 2003. Data caches in multitasking hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. 154–166.

X. Vera, B. Lisper, and J. Xue. 2007. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems* 7, 1, 1–38.

S. A. Ward and R. H. Halstead. 2002. *Computation Structures*. Kluwer Academics.

R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*. 192–202.

J. Whitham and N. Audsley. 2010. Studying the applicability of the scratchpad memory management unit. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. 205–214.

M. E. Wolf and M. S. Lam. 1991. A data locality optimizing algorithm. *ACM SIGPLAN Notices* 26, 30–44.

J. Xue and X. Vera. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Transactions on Computers* 53, 5, 547–566.

Received June 2012; revised September 2013; accepted December 2013