A small and effective data cache for real-time multitasking systems

Juan Segarra*, Clemente Rodríguez†, Rubén Gran*, Luis C. Aparicio* and Víctor Viñals*
*Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza, España
*Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, España
†Dpt. Arquitectura y Tecnología de Computadores, Universidad del País Vasco, España
Email: jsegarra@unizar.es, acprolac@ehu.es, rgran@unizar.es, luisapa@unizar.es, victor@unizar.es

***Abstract*—In multitasking real-time systems, the 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 with data caches. In this paper, 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. Such results translate into doubling the performance of the tested real-time multitasking experiments, which (increasing from 75 to 89%) approaches the ideal case of always hitting in instruction and data caches. Additionally, we show that using partitioning on our proposed hardware only provides marginal benefits.**

I. 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, in order 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, in general, it is necessary to know the previous access sequence in order to calculate the latency of a given access in advance. Resolving these *intra-task* interferences is a difficult problem in its own right. Moreover, real-time systems usually work with several tasks which may interrupt each other at any time. This makes the problem much more complex, since the cost of *inter-task* interferences must also be identified and bounded. Furthermore, both 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 paper 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 lines. These data cache lines can only be replaced by their associated instructions, i.e., only such instructions have data cache replacement permission. Since each memory instruction replaces its own data cache line, it prevents pollution 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 calculated using data reuse theory [1]. Next, any WCET optimization method can be used to decide which instructions have such permissions, depending on the cache size, the inter-task interferences, etc. To obtain such instructions, we extend the Lock-MS WCET analysis method [2], [3].

Compared to a conventional data cache, the novel features of our data cache design are the following: i) Achievement of *high performance* with a *small* cache size (256 bytes), *independently of program data size*. Contents are replaced in a similar way to in conventional caches, maintaining its dynamic behavior. ii) *Predictability* and *no pollution*. Only selected instructions can replace data cache lines. This is achieved by indexed-based replacement, with the advantage that the usual replacement overhead (e.g. LRU) is eliminated. iii) The number of misses is calculated using data reuse theory as developed for conventional caches [1]. This enables references to unknown addresses (e.g., pointers) to be analyzed, which is not possible with other methods (e.g., [4], [5]).

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 [6].

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).

The rest of this paper is organized as follows. Section II reports related background on data caches. Our proposed data cache is described in Section III. Section IV shows how to specify its behavior as an ILP model. Sections V and VI describe the experimentation environment and the results obtained. Finally, Section VII presents our conclusions.

II. Related work

Data caching is much more complex than instruction caching, since references may present very different behaviors: scalar vs. non-scalar, global vs. local, activation blocks (i.e. subroutine context), dynamic memory, etc. 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*. So, 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 [7]. The instruction address of the memory access (load/store) and a hardware predictor allow to predict the type of locality. The size of such caches is higher than 8 KB and they have no pollution for references to non-scalar variables. Early approaches focused on a stack cache [8]. Other authors have suggested hardware modification to include a register-based structure for storing part of the activation block [9]. There are also proposals to store accesses to the heap in a small cache (2 KB) or a large cache (32 KB) [10]. Finally, one study proposed three caches to manage the three memory regions (stack, global and heap) [11]. Additionally, this avoids conflicts between regions and provides the required size for each: small for the stack and global, and large for the heap.

Locking data caches and scratchpad memories are alternatives intended to capture temporal locality and avoid pollution [12], [13]. 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 run time [6], [14]. 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 [15], based on the data reuse theory for LRU replacement in conventional data caches [1]. So, if preloading and locking the data cache with a given selection of data reduces the number of misses, the required 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 be also locked to reduce pollution. This technique is particularly sensitive to optimizations that increase locality (*padding* and *tiling*), and it may be combined with partitioning techniques to avoid inter-task interferences (e.g., [16]).

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. Lastly, modification of task code is not required and optimizations are not so important.

III. ACDC cache

Our proposed data cache is able to take advantage of temporal and spatial locality. Usually, data caches are *data-driven*, i.e. 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, i.e. on *how* the data are accessed.

A. Hardware description

Our proposed data cache structure, ACDC (Address-Cache/Data-Cache), is a small fully-associative data cache (DC) with an instruction driven replacement policy. We assume a write-back policy, since it involves less traffic than write-through. Since the replacement policy is external to the DC, no replacement mechanism is needed unlike in other conventional caches (e.g., LRU, FIFO, etc.). So, 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 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 (read or write) data is serviced/updated directly from main memory. Since each instruction with replacement permission is only allowed to replace a predefined data cache line for its references, there is no pollution.

This behavior can be implemented in many ways. We describe an implementation (Figure 1) that does not modify the instruction set and only requires the data cache to be fully associative with replacements disabled. Accordingly, the only requirement is a new hardware structure to manage the instruction replacement permissions. This is simply a table that stores the memory instruction addresses (PC) which can replace the cache lines, and the specific data cache line that they can replace. For instance, this table or address cache (AC) could be implemented by a small fully-associative lockable cache, not requiring the conventional replacement capabilities, where the tag is the memory instruction address and its entry points to the specific data cache line that this instruction is allowed to replace.

In context switches, the task starting/continuing the execution stores in the AC its own set of instruction addresses with replacement permission and the index of the DC line they are

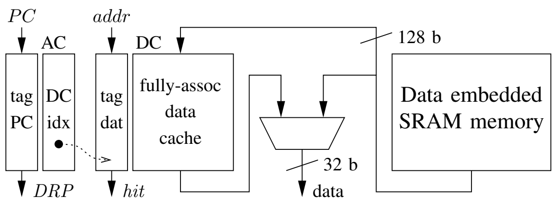


Figure 1. Data memory architecture. AC contains addresses of instructions with data cache replacement permission and the index to their associated DC line

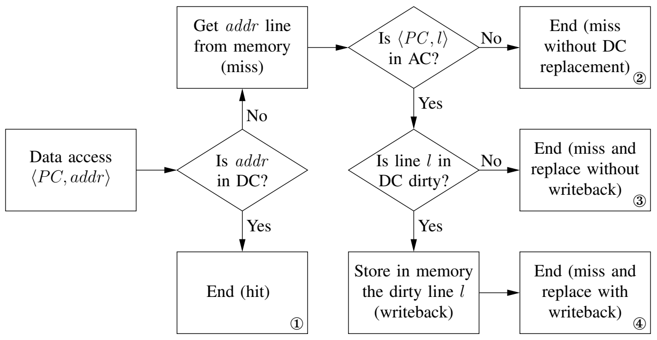

Data access $\langle PC, addr\rangle$
Is $addr$ in DC?
No
Yes
Get $addr$ line from memory (miss)
End (hit) ①
Is $\langle PC, l\rangle$ in AC?
No
Yes
End (miss without DC replacement) ②
Is line l in DC dirty?
No
Yes
End (miss and replace without writeback) ③
Store in memory the dirty line l (writeback)
End (miss and replace with writeback) ④

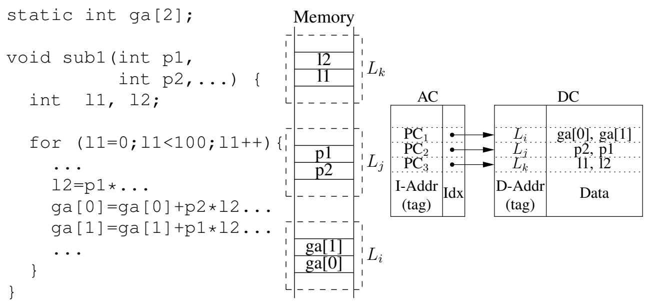
Figure 2. Data access procedure

allowed to replace. Then, data accesses are made as depicted in Figure 2. The DC is accessed with the desired data address ($addr$). On hits, it reads or writes the requested data and the access procedure is completed (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 misses in the AC, the DC will not be replaced (②). On AC hits, the data access has replacement permission, and the DC line to replace is indicated by the index stored in the AC. In this case, having a write-back policy, if the currently stored line is dirty, it will be written back to memory (④). Otherwise, the DC will be replaced without write back (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.

B. 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 into data locality, in this paper we consider only the most common cases. Such cases cover most situations and can be modeled in a simple way.

Temporal reuse: Memory references present temporal reuse when they access the same memory address repeatedly during program execution. In this paper, only accesses to



```
static int ga[2];

void sub1(int p1,
          int p2,...) {
  int  l1, l2;

  for (l1=0;l1<100;l1++){
    ...
    l2=p1*...
    ga[0]=ga[0]+p2*l2...
    ga[1]=ga[1]+p1*l2...
    ...
  }
}
```

Figure 3. Example of temporal locality in code, memory and the ACDC structure

scalar variables are considered for temporal reuse. Although, in general, array reuse is better described by spatial reuse (see below), we consider small arrays that fit in a single cache line as if they were scalar variables. When such 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 [1] 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 3 shows an example of the temporal reuse of different structures, namely global (static) variables, local variables (within the function scope) and function parameters. For variables not stored in registers, their location in memory (in specific memory lines L) will determine their associated temporal reuse. In order to exploit such reuse, the first reference to each of these memory lines will be given replacement permission in order that they can be cached, i.e., 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), since all references to this variable will access the same memory line. Furthermore, assuming that the function `sub1` always works with the same stack addresses (roughly speaking, it is always called from the same nesting level), all its accesses will have temporal locality, with a single miss on the first access to each memory line. If `sub1` is called from n different stack addresses, there will be n misses for each line with temporal locality instead of a single one.

Spatial reuse: There is spatial reuse of memory references when they access close memory addresses during program execution [1]. In this paper, only sequential accesses (those with *stride* 1) are considered, since other strides are unusual in real-time applications and formulae would become less clear. Nevertheless, considering other constant strides would be trivial. As above, 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 4) using our proposed ACDC structure. We present three cases, namely, the matrix multiplication

```
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

Figure 4. Matrix multiplication algorithms

Table I
DATA REFERENCES IN MATRIX MULTIPLICATION (FIGURE 4)

NonOpt	T S G	AC DC	Access	Miss	WrBk
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**	**Access**	**Miss**	**WrBk**
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
Total Opt1 − NonOpt:			$-2n^3+2n^2$	0	0
Opt2	**T S G**	**AC DC**	**Access**	**Miss**	**WrBk**
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
Total Opt2 − NonOpt:			$-n^3+n^2$	$-n^3+n^3/b$	$\frac{n^3-n^2}{b}$

code of the *matmul* benchmark (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. In order to simplify the mathematical notation, we assume $n \times n$ aligned matrices, with n being a multiple of the cache line size b. Further, we place the write back cases in the instruction which 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, i.e., 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. 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 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 method for analysis and minimization of WCET.

IV. LOCK-MS EXTENSION

The aim of Lock-MS is to identify a selection of instruction lines from memory such that, when locked into the instruction cache (IC), the schedulability of the whole system is maximized [2], [3]. 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. 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. This selection has the drawback of being reloaded in the AC on each context switch, and the content in the DC must be considered flushed.

Previous work on Lock-MS has grouped all costs of a given instruction memory line into 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 [2], [3].

A. 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 position of the element accessed in the array (based on the loop iteration). The association $\langle pc, ref\rangle$ cannot change, i.e., a given memory instruction pc always accesses memory using a given reference ref. However, several memory instructions reusing the same data structure will 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 IV-C.

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.

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

So, the additional (ACDC) costs for each context switch would be those due to preloading the AC and those due to assuming that the DC has been flushed. This value times the maximum possible number of context switches in Rate-Monotonic scheduling is added to the WCET.

$$\begin{aligned} dataSwitchCost_i = \; & ACpreloadCost \cdot nInsDRP_i + \\ & DCoverestCost \cdot nRefsDRP_i \end{aligned}$$

B. 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, data cache misses (with or without replacement) and line write-backs (see Figure 2). Since our proposed method of data cache management is instruction-based, 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 (eq. 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 [2].

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

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 (e.g., see Table I).

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

Further, the number of line write-backs depends on whether all the instructions using the accessed data are loads or not. If all of them are loads, the number of write-backs is clearly 0, since 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, i.e., 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).

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

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, while those without DRP will always ($nfetch$) miss. Hence, the resulting constraints are:

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

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 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.

C. Determine the number of ACDC misses

In order to determine the number of ACDC misses, we distinguish between references to scalar variables and to non-scalar 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 $nAct_{sub}$ distinct activation block addresses for their associated subroutine sub, the number of misses is $nAct_{sub}$.

In order to calculate the number of misses of non-scalar variables, we consider serial loop nest data reuse and locality theory, briefly introduced below [1]. Each iteration in the loop nest corresponds to a node in the *iteration space*. In a loop nest of depth n, this node is a vector space in n dimensions, and is identified by its index vector $\vec{p} = (p_1, p_2, \ldots, p_n)$ where p_i is the loop index of the ith 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 of the loop of index k of $A[in_1, in_2, \ldots, in_k, \ldots, in_d]$. 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 i_1 and 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, since 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, i.e., 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. That is, 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.

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 iff $\vec{c_1} = \vec{c_2}$, i.e., if both memory references are identical. One such reference will be SSR also, so we classify it as SSR.

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 non-scalar 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 5 shows how to obtain the number of misses for non-scalar 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.

D. Structure-Based ILP

Previous constraints correspond to a path-based ILP model, since they specify all variables with subindexes for the path j in the task i. We use such notation in order to make them easier to understand. In general, however, the path information is not relevant, since 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 [2].

In order 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 as with only one path with serial locality. That is, all alternative

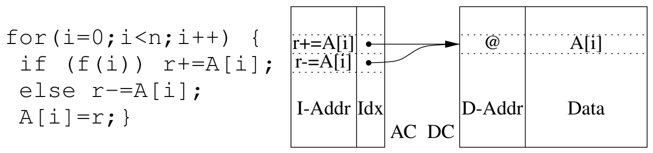
Table II
REUSE MATRICES OF MATRIX MULTIPLICATION (FIGURE 4, TABLE I)

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	$\frac{n^3}{b}$

```
Require: $H, memLines(A)$  # transformation matrix, data structure size
Ensure: $ACDCmisses$
 1: if $e_n \in Ker(H_s)$ then                    # 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 $\exists$ 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      # $\forall$ 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.

Figure 5. Algorithm to get $ACDCmisses$ for STR/SSR non-scalar refs



```
for(i=0;i<n;i++) {
 if (f(i)) r+=A[i];
 else r-=A[i];
 A[i]=r;}
```

Figure 6. Simple example of *cooperative* reuse involving different paths

Table III
TASK SETS "SMALL" AND "MEDIUM" WITH DATA ACCESS INFORMATION

	Task	Dir-mem WCET	Period	% dat acc.	Temp (%)		Spa (%)		% ca. data	DC lines
					Self	*Gr.*	*Self*	*Gr.*		
small	jfdctint	18808	40432	17.6	33.8	0	18.5	40.7	93.06	11
	crc	213221	478560	13.6	41.0	27.4	15.9	9.0	93.29	6
	matmul	834359	2169448	29.8	0	0	51.2	23.8	74.94	5
	integral	1587329	6534486	42.1	13.3	86.6	0	0	99.91	1
medium	minver	16793	43902	33.0	9.3	25.4	28.7	5.4	68.88	16
	qurt	21644	61908	38.2	13.8	74.1	0.5	1.1	89.54	10
	jfdctint	18808	62066	17.6	33.8	0	18.5	40.7	93.06	11
	fft	4742498	14792660	22.9	11.4	75.7	5.7	0	92.82	12

paths have equivalent memory instructions accessing the same references. This means that different instructions may have data replacement permission on the same data cache line. However, since these instructions are equivalent, they work *cooperatively* and do not pollute each others 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 6 shows a simple example (references to array `A`) of this situation.

V. EXPERIMENTATION ENVIRONMENT

All 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 constraints follow the Lock-MS model to minimize the WCET [2]. The feasibility of such a system can be tested in a number of ways [17]. *Response time* analysis is one of these mathematical approaches, and it is used as our main multi-task metric.

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*. The "small" and "medium" task sets have been used in previous studies with similar periods [2], [12]. 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. The option -O2 implies *fomit-frame-pointer* (the frame pointer is not used). Also, the stack can grow only once in each routine (*Stack Move Once* policy).

The WCET in Table III refers to a system without caches or buffers (i.e., direct access to separate 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. The remaining columns in this table are discussed in Section VI.

Table IV
TIMING (CYCLES) CONSIDERED IN OPERATIONS (SEE FIGURE 2)

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 repl. without wb	1+1+6 (addr. + ACDC + memory)
④ ACDC miss and repl. with wb	1+1+6+6 (ACDC miss + memory)

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 1 KB (64 sets), all direct mapped. All the 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 [2], [3], [12]. It is important to note that a careful modeling of the instruction supply is essential in order to study the real impact of the ACDC on the system, since 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. In order to compute memory circuit delays we have used Cacti v6.0 [18], a memory circuit modeling tool, assuming an embedded processor built in 32 nm technology and running at a processor cycle equivalent to 36 FO4[1]. All the 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 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 4 times better than a data miss. That is, we assume a base system with a very good performance in order to truly test our ACDC. Other studies consider off-chip main memories and assume a higher miss penalty, such as 38 cycles [14] or 64 cycles [9]. Clearly, in these systems the room for improvement is much higher. Thus, our results can be seen as a lower bound on performance, which would rise if the T_{miss}/T_{hit} ratio were increased.

[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 32 nm technology would result in a clock frequency of around 2.4 GHz, which is in line with the market trends [19].

VI. Results

In this section we combine our proposed data cache structure with the instruction fetch components (lockable instruction cache, line buffer and instruction prefetch) [2], [3]. Figure 7 shows the response time speedups relative to a baseline system with an instruction LB but no caches (first bar of group 0). As a reference, the baseline 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 first bar group (labeled 0) assumes no IC, and the remaining four bar groups vary the IC size from 64 to 1024 bytes. The ACDC size is not varied since, even with such small size, tasks use only a subset of its lines. The IC is a dynamically locked instruction cache with contents selected using the Lock-MS method [2], [3]. First bar (LB+DC Lock-MU) represents a system with an LB and a statically locked data cache using the Lock-MU method on data [12]. 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 below.

A. 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) are much more dependent 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 caches. Also, 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 speedup 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 speedup is roughly 1.5.

B. ACDC analysis

As outlined above, across all considered memory configurations and task sets, ACDC enhances performance by a factor of 1.5 to 2. In order 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 accesses which have permission to be placed in the DC (column *% ca. data*) 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. As studied above (see Table I), it

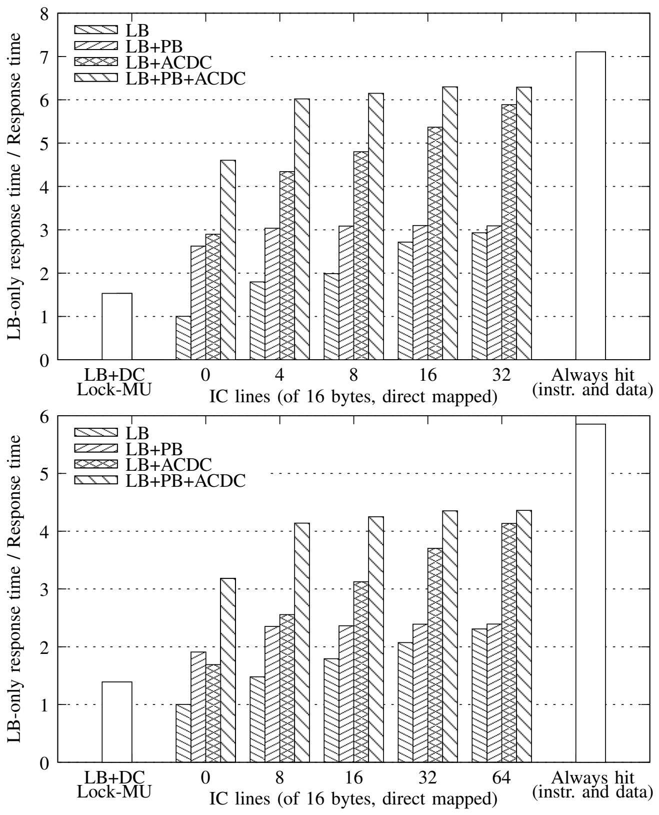


Figure 7. Response time speed-up (normalized to a cacheless LB-only system) of several memory configurations for the small (upper) and medium (lower) task set

accesses a matrix by columns (stride > 1), which prevents any small data cache 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 above, 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%), since 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 just 16 lines in AC and DC, i.e., with a DC of 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 6 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. 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). Thus, our proposed data cache is very suitable for embedded systems, where size is usually a key factor.

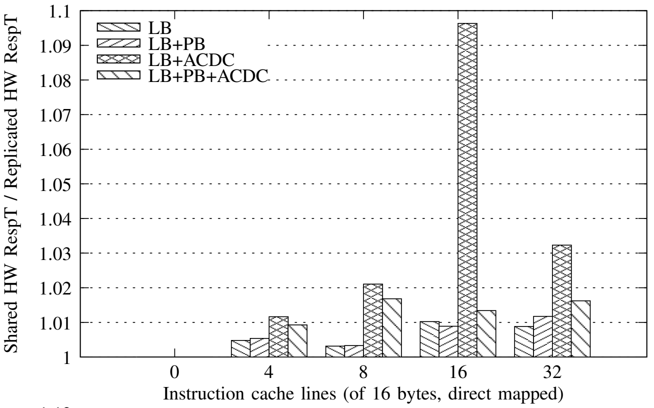
Additionally, Table III shows in column 5 (% dat acc.) the percentage of data accesses over the total memory accesses (data+fetch) for each task. It can be seen that between 13% and 42% of memory accesses are data accesses, which represents an important factor in the WCET calculation. Columns 6 and 7 (Temp and Spa) show the percentage of data accesses managed by ACDC, divided into self or 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 separate temporal and spatial localities.

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 [14]. 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 [14]. 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, i.e., specifying different data replacement permissions (AC contents) for different task stages.

C. Partitioning

In this section we analyze how much improvement would come from partitioning resources, i.e., without preloads on context switches. To focus in the effects of inter-task 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. That is, any partitioning technique will perform as well as or less well than a per-task hardware replication.

Figure 8 shows the same memory configurations as above. In this case, the vertical axis shows the response time speedup due to full replication (shared/replicated). That is, values of the response time with shared hardware include preloading costs and overestimations due to inter-task interferences, 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, i.e., lines that were not cached because their associated penalties on context switches were too large, can be cached now. Nevertheless, Figure 8 shows very marginal benefits when replicating hardware. Both for the small and medium task sets, results show improvements of less than or



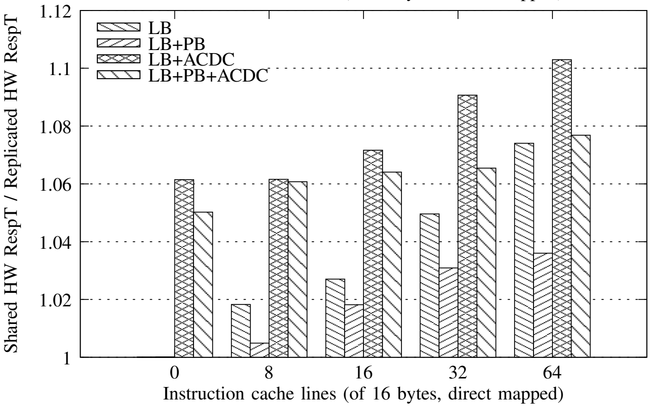


Figure 8. Response time speedup of full resource per-task replication of several memory configurations for the small (upper) and medium (lower) task set

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 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 above, replication offers the highest performance bound on partitioning, since 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 into account that replicating hardware implies considerable economic costs, it seems that such marginal benefits are not worth replication or partitioning.

VII. Conclusions

In this paper, 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 instruction replaces its own data cache line, pollution is avoided and performance is independent of the size of the data structures in tasks. Moreover, by its nature, it guarantees that data cache lines cannot be replicated. This

ACDC is relatively easy to analyze (similar level of difficulty to a locked instruction cache) but maintains its dynamic behavior, since contents are replaced in a similar way to in conventional caches. Further, since the selection of instructions with replacement permission uses 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 our proposed ACDC component. In order to study the real impact of the ACDC in the system, experiments included 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 reached 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, 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.

Finally, we repeated our experiments assuming that each task had its own replicated hardware in order 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.

References

[1] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, pp. 30–44, May 1991.

[2] 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," *Journal of Systems Architecture*, vol. 57, pp. 695–706, 2011.

[3] ——, "Combining prefetch with instruction cache locking in multitasking real-time systems," in *Proc. of the IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications*, Aug. 2010, pp. 319–328.

[4] Y. T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: beyond direct mapped instruction caches," in *Proc. of the IEEE Real-Time Systems Symposium*, Dec. 1996, pp. 254–264.

[5] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon, "Timing analysis for data caches and set-associative caches," in *Proc. of the IEEE Real-Time Technology and Applications Symposium*, Jun. 1997, pp. 192–202.

[6] J. Whitham and N. Audsley, "Studying the applicability of the scratchpad memory management unit," in *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010, pp. 205–214.

[7] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *Proc. of the Int. Conf. on Supercomputing*, 1995, pp. 338–347.

[8] S. A. Ward and R. H. Halstead, *Computation Structures. 2002*. Kluwer Academics, 2002.

[9] R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, and F. Tirado, "Stack filter: Reducing L1 data cache power consumption," *Journal of Systems Architecture*, vol. 56, no. 12, pp. 685 – 695, 2010.

[10] M. Geiger, S. McKee, and G. Tyson, "Beyond basic region caching: Specializing cache structures for high performance and energy conservation," in *Proc. on the Int. Conf. on High-Performance and Embedded Architectures and Compilers*, 2005, pp. 102–115.

[11] H.-H. S. Lee and G. S. Tyson, "Region-based caching: an energy-delay efficient memory architecture for embedded processors," in *Proc. of the Int. Conf. on Compilers, architecture, and synthesis for embedded systems*, 2000, pp. 120–127.

[12] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. of the IEEE Real-Time Systems Symp.*, Dec. 2002.

[13] I. Puaut and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison," in *Proc. of the Design, Automation Test in Europe Conference Exhibition*, Apr. 2007, pp. 1–6.

[14] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *Proc. of the IEEE Real-Time Systems Symp.*, Dec. 2003, pp. 154–166.

[15] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 703–746, 1999.

[16] R. Reddy and P. Petrov, "Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems," in *Proc. of the Int. Conf. on Compilers, architecture, and synthesis for embedded systems*, 2007, pp. 198–207.

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

[18] N. Muralimanohar, T. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," University of Utah and Hewlett Packard Laboratories, Tech. Rep., 2007.

[19] "Chart watch: High-performance embedded processor cores," *Microprocessor Report*, vol. 22, pp. 26–27, Mar. 2008.