A predictable hardware to exploit temporal reuse in real-time and embedded systems[1][2]

R. Gran[a,b], J. Segarra[a,b], A. Pedro-Zapater[a], L. C. Aparicio[a,b], V. Viñals[a,b], C. Rodríguez[c]

[a] *Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza, España*
[b] *Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, España*
[c] *Dpt. Arquitectura y Tecnología de Computadores, Universidad del País Vasco, España*
HiPEAC-3 NoE (European FP7/ICT 287759)
{rgran,jsegarra,albapz,luisapa,victor}@unizar.es, acprolac@ehu.es

Abstract

In this paper we propose a new hardware data cache (FAFB, fully-associative FIFO tagged buffers) to complement the data cache in processors. It provides predictability when exploiting temporal reuse in array data structures, i.e. it allows an accurate WCET analysis, which is required in real-time systems. With our hardware proposal, compiler transformations that exploit such reuse (essentially tiling) can be safely applied. Moreover, our proposal has other features of particular interest to embedded systems, where a set of well-tuned applications run in a hardware platform which may be constrained in size, complexity and energy consumption. In order to test the most uncommon features of the FAFBs (predictability and effectiveness with a small size), we perform a worst-case analysis on several kernel algorithms for embedded and real-time computing, showing the interaction between tiling and our hardware architecture. Our results show that the number of data cache misses is reduced between 1.3 and 19 times on such algorithms.

Keywords: WCET, data cache, tiling, real-time

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, in order to improve performance, current processors perform many operations with a variable duration. In general, the cache hierarchy is the component with the largest impact to the WCET, both due to its continuous operation and its variable latency. Conventional caches offer a very good performance in average, but real-time systems require predictability in the worst case, i.e. for a given memory access, provide the number of processor cycles that will take such access to be completed. Essentially, this means to be able to know the cache hits and misses previously to the execution, but such analysis has an exponential cost [1].

Data caching analysis in real-time systems is much more complex than instruction caching, since it adds two important difficulties. The first one is that a given memory instruction may access different data addresses during execution. The second one is that data addresses may be unknown at compile time, e.g. accessing memory through unknown indexes, pointers, or *hash* functions. Hence, although analyzing conventional data caches is theoretically possible [2, 3], its practical use has limitations. Such difficulties

[1] This work was supported in part by grants TIN2013-46957-C2-1-P and Consolider NoE TIN2014-52608-REDC (Spanish Gov.), and gaZ: T48 research group (Aragón Gov. and European ESF).

imply that WCET analysis of algorithms that work with large data structures is hard, and the obtained WCET may have unacceptable overestimations.

In order to facilitate the WCET analysis considering data caching, locked data caches and scratchpad memories may be used. The content in such components is fixed and controlled by software, so its behavior is easily predictable. However, the complexity now deals with the discovery of a content-selection policy able to obtain a low WCET [4, 5]. Note that even the best selection of contents may result in a WCET worse than that of a conventional cache, since locked caches and scratchpads lack the dynamic reuse exploitation of conventional caches. Trying to exploit such dynamism, some authors just lock the data cache when the predictability analysis is complex (e.g. on array walks in loops) or when they want to preserve the cache contents from data references to unknown addresses [6]. Another option is to manage the locked cache as a software-controlled prefetch buffer, where specific contents are loaded before being accessed [7]. However, proposals of dynamically changing the loaded contents as the program executes (dynamic locking methods) must add instructions for managing such changes into the task. Thus, the potential WCET improvement decreases with these added instructions.

Finally, it is worth mentioning two recent data cache designs specifically targeted at real-time systems. The first one is a cache with random mapping and replacement [8]. Such cache enables probabilistic WCET computation, that is, compute a WCET value with a probability of underestimation (i.e. unsafety) lower than, for instance, the probability of hardware failure. Another novel cache design is the ACDC [9]. It is composed of a very small fully-associative data cache, whose contents can only be replaced by a preselected set of instructions. Since each preselected memory instruction replaces its own data cache line, ACDC provides a predictable behavior and it is easy to analyze and optimize for WCET minimization. However, it does not allow to exploit temporal reuse in array data structures. This drawback in the ACDC seems hard to avoid, since adding the associativity required to exploit such temporal reuse would make the ACDC behavior much harder to predict.

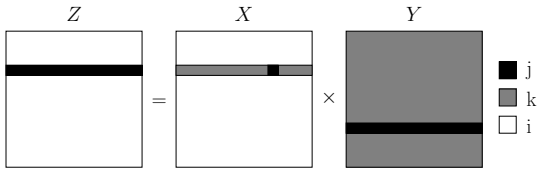
In this paper we propose a small data cache specially targeted at exploiting temporal reuse in array data structures. It is designed to work coupled with any L1 data cache, so that our proposed hardware catches temporal locality and at the same time prevents possible conflicts in the coupled data cache. Our proposal consists of a series of very small fully-associative FIFO tagged buffers (FAFBs), each one associated to a specific load/store instruction. In this way, each FAFB is able to cache a small subset (tile) of a particular data structure without conflicts with other data structures.

Our proposed hardware has the following features:

- *Small* size: 2 FAFBs with 4 cache lines per FAFB are proposed, totaling 83 bytes. Using our hardware adequately, reuse exploitation is *independent of the size of the data structures*.
- *No conflicts*. Only selected instructions can replace lines in FAFBs and no other instruction competes for storing data in such lines. This translates into *i*) no pollution (no other instruction can replace the contents cached in the FAFBs by the selected instructions) and *ii*) predictability (the number of hits and misses in the worst case can be easily calculated by using the well known data reuse theory [10]).
- *Easily exploitable through compiler transformations*. Some code transformations (essentially tiling) can easily exploit the temporal locality usable by FAFBs.
- *Energetically efficient*. Since the compiler knows when each FAFB will be used, they could be enabled/disabled on request of the running tasks.
- *Specially suitable for embedded devices*, where size, production cost, and energy consumption are key factors.
- *Specially suitable for hard/soft real-time systems*, where worst-case predictability is the key factor.

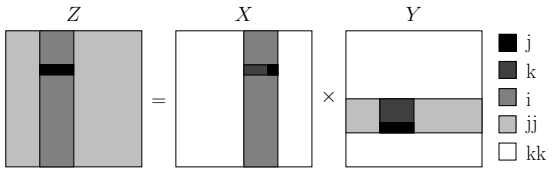
All these features allow *i*) an efficient execution of complex algorithms (both in average and worst case) by exploiting temporal reuse on array data structures, and *ii*) an easy and accurate WCET analysis of such algorithms, which is hard with current existing hardware.

```
for (i=0; i<N; i++)                    // N iterations
  for (k=0; k <N; k++) {               // N iterations
    r = x[i][k];
    for (j=0; j<N; j++)                // N iterations
      z[i][j]=z[i][j]+(r*y[k][j]);
  }/* (store) (load) */
```



(a) Matrix multiplication code and its data access pattern.

```
for (kk=0; kk<N; kk+=B)                // N/B it
  for (jj=0; jj<N; jj+=L)              // N/L it
    for (i=0; i<N; i++)                  // N it
      for (k=kk; k<min(kk+B,N); k++){    //B it
        r = x[i][k];
        for (j=jj; j<min(jj+L,N);j++)    //L it
          z[i][j]=z[i][j]+(r*y[k][j]);
      }/* (store) (load) */
```



(b) Tiled version of matrix multiplication and its data access pattern.

Figure 1: Code and data access pattern in (a) non-tiled and (b) tiled matrix multiplication [11].

The rest of this paper is organized as follows. Section 2 describes the problem of temporal data reuse. In Section 3 we describe our hardware proposal. Section 4 details the evaluation environment and parameters. Section 5 shows the obtained results focusing on real-time and embedded systems. Finally, Section 6 presents our conclusions.

2. Exploitation of temporal data reuse and tiling

Data reuse is a common property of computer programs, and there are many studies on how to benefit from it. Essentially, such benefit comes from caching the data to be reused, so that they remain "local" when they are used again. Thus, exploiting data reuse reduces execution time and energy consumption, since a first-level cache hit is faster and consumes far less energy than a next-level hit or main memory access.

Perhaps one of the most tricky access patterns to exploit is the temporal reuse on large array data structures, because caches may result ineffective at exploiting it [10]. For instance, assuming temporal reuse by walking multiple times a data array larger than a cache level, it results in no temporal locality hits for conventional replacement policies (LRU, NRU, FIFO,...), since the data accesses evict their own cached data before being reused. In order to overcome this limitation, there are extensive studies on *tiling/blocking* compiler transformations [10, 11, 12]. Instead of working with large data structures, these transformations modify the algorithm to work with small chunks (tiles) of those large data structures. That is, they make tiles of the initial data structure and process such tiles one after another. In this way each one of these small tiles fits in cache, which effectively exploits the temporal locality. Figure 1 shows a representation of this transformation on the matrix multiplication algorithm, as shown in [11]. It represents the three loops required for a typical non-tiled matrix multiplication (a), and its tiled version with five loops (b). The non-tiled version uses the loop j to walk horizontally matrices Z and Y, and the tiled version has divided this loop into two loops j and jj, where the j loop walks horizontally inside *small tiles* in matrices Z and Y, and the jj loop moves these tiles horizontally. Similarly, the k loop in the non-tiled version walks horizontally the matrix X, whereas the tiled version uses the k loop for walking horizontally inside *small tiles* in the X matrix, and the kk loop moves these tiles horizontally. Working with such tiles, data accesses present a closer temporal reuse (i.e. the number of accesses between two accesses to the same data address is smaller), which allows a better exploitation of the temporal locality. Tiling of perfectly nested loops is implemented by some compilers and tools like PLUTO [13]. Also, tiling can be applied on imperfectly nested loops [14, 15, 16].

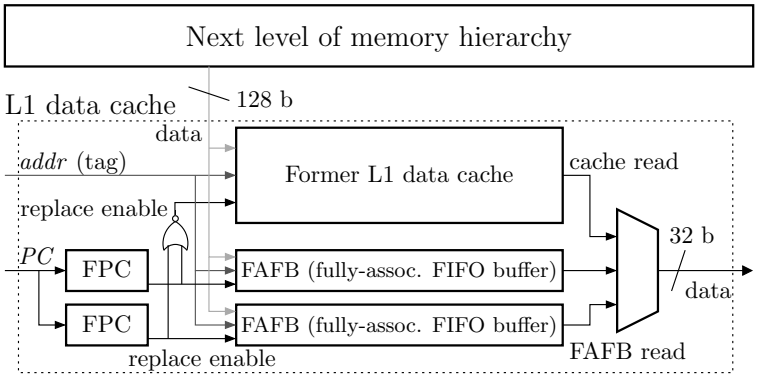


Figure 2: First-level data memory hierarchy with two FAFBs.

Many algorithms used in fields such as robotics, signal processing, image processing, communications systems, cryptography, computer vision, and adaptive control systems, can be exploited by tiling transformations [17, 18, 19, 20, 21]. Obviously, the achieved locality depends on cache parameters (size, line size, set-associativity, etc.), cache policies (replacement policy, write policy, presence of coupled victim cache, locked sets/ways/lines, etc.), hardware clients of the cache (instruction/data/unified in single-processor, private/shared in multi-processor, prefetching, etc.), and software clients of the cache (singletask/multitask/multithread/parallel execution). Thus, applying a tiling transformation is not straightforward, since the tile size to use will depend on all these factors. Moreover, even the most simple algorithms work with several data structures, with their inherent cache conflicts. This means that transforming an algorithm to optimize the accesses to a specific data structure may have an adverse effect on the others, and even on the optimized structure itself due to the cache conflicts, harming the global performance. Hence, the decision to apply tiling is not straightforward either, and a key challenge is to provide an homogeneous and effective hardware support for the tiling transformations.

3. FAFBs description

In this section, we describe a new hardware structure that exploits temporal reuse of algorithms in array data structures, specially when using tiling transformations. Moreover, its behavior is *predictable*, i.e. the number of hits and misses of a given task in the worst case can be easily and accurately calculated previously to its execution.

Figure 2 shows the first level of our data memory hierarchy (dashed box in the figure). It consists of two fully-associative FIFO tagged buffers (FAFBs) intended for exploiting temporal reuse (although more FAFBs could be used) coupled with a former data cache, which would exploit other reuse types. Each FAFB stores B memory lines. Associated to each FAFB, an FPC register holds the PC of the memory instruction which has replacement permission in that FAFB, i.e. the single particular load/store that may replace these B memory lines. The former L1 data cache can be any type of cache, for instance a conventional LRU cache memory, a scratchpad memory, or an ACDC.

FAFBs work as follows (Figure 3). After computing the data memory address, loads and stores look up the former data cache and the FAFBs in parallel. In case of hit in any of such structures, the data line is serviced from the hitting hardware structure, either data cache or FAFBs, ①. In case of miss, it must be determined where it must be cached. So, the PCs stored in the FPCs are compared with that of the current memory instruction (PC). If there is no match in FPCs, the former cache manages the rest of the process, i.e. caching the data line, possibly replacing an old one and servicing the requested address, ②. Otherwise, the line will be cached in the matching FAFB, ③. In this case, the FIFO replacement policy determines the cache line to be replaced in the FAFB. Having a write-back policy, if the victim line is dirty, it will be written back to memory. Otherwise, the FAFB will be refilled without writing back.

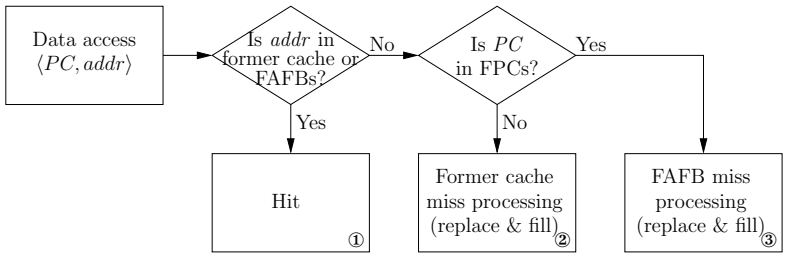


Figure 3: Former data cache plus FAFBs access procedure.

To exploit the FAFBs, the memory instruction with replacement permission of each FAFB (the one whose PC is in FPC) should have a memory access pattern with temporal reuse [10]. In order to set values in FPCs we assume they are mapped in memory similarly to registers of memory mapped I/O devices. Therefore, no change to the instruction set architecture is needed, and regular store instructions may be used to set these values. Ideally, the compiler should select the particular instruction PCs (FPCs) when applying tiling transformations or analyzing data reuse, although an external analysis/profiling or even the programmer could select them too. Such PCs would be reloaded into the FPCs at context switches.

It is important to note that any type of cache could be coupled with our FAFBs. Also, depending on whether the desired target is speed or energy efficiency, both the coupled data cache and the FAFBs could be looked up in parallel, or we can only lookup the data cache *after* a FAFBs miss. By having parallel lookup, FAFBs add no latency. Alternatively, a sequential lookup could take advantage of the specific phases of code in which FAFBs exploit temporal locality extensively. In such phases most hits will come from FAFBs, avoiding activity in the coupled cache. Locating such phases can be done by the compiler when it tiles the code and configures FPCs. Out of these phases, FAFBs would be disabled to avoid lookup latency and energy consumption.

Thus, by construction, our proposed hardware caches the data referenced by specific memory instructions, freeing the coupled data cache from this duty and avoiding the possible pollution of such references in the coupled data cache. Also, by construction, it is not possible to have duplicated data lines both in the data cache and the FAFBs, since replacements are only triggered when misses occur in all the structures, and later the missing data line is directed to a single destination which depends on the FPCs contents.

Finally, let us highlight that FAFBs (even if coupled with an ACDC) are not equivalent in terms of predictability to an ACDC with an extended associativity per DC entry (that is, an ACDC where each AC entry points to a DC entry with multiple cache lines managed with some replacement policy. Whereas both the ACDC and FAFBs are easily predictable, an extended ACDC with several cache lines per DC entry is not. This means that the worst case of such an extended ACDC cannot be analyzed by the proposed theory [9].

4. Evaluation environment

In this section we analyze the parameters and sizing required for our proposal. Also, we define the different systems that are evaluated below.

Since FAFBs can be sized to adapt to specific problems we describe them through the parameters B (number of cache lines, or entries, per FAFB) and L (number of elements per cache line). For instance, a FAFB with $B = 4$ and $L = 4$ integer elements of 4 bytes each, stores 16 elements in 64 bytes, and requires four comparators to perform an associative lookup; as another example, a FAFB of 64 bytes with $B = 2$ and $L = 4$ elements of 8 bytes each could hold tiles of 8 double-precision floating-point numbers. So, for an algorithm working with tiles with temporal reuse traversed sequentially (element stride is one), the allocated FAFB will be used in its entirety. In case of tiles with temporal reuse of sequences with larger strides (e.g. walking repeatedly the columns of a row-major ordered matrix fitted into a tile), temporal locality

could only be exploited for a single element in each FAFB entry. In such cases, temporal reuse would be limited to sequences of B elements, no matter the element size.

As a reference, assuming FAFBs with $B = 4$ cache lines of 16 bytes each, and an addressable space of 1 GiB (30 address bits), the following control information is required. Each FAFB must store the program counter (PC) address of the memory instruction which has replacement permission on this FAFB. Assuming aligned instructions of 4 bytes, 28 bits are required for a PC. Also, each entry (of 16 bytes) in a FAFB must store the corresponding tag plus two state bits (valid and dirty), that is, 26+2 bits. Finally, the FIFO replacement system must point to one of the four ($B = 4$) entries, that is, 2 bits. Thus, one FAFB, with a PC and four entries consisting of tag, state, data, and FIFO replacement, would use 664 bits (83 bytes). This configuration will be used in Section 5, along with a discussion on parameter sensitivity.

As explained, by construction, FAFBs exploit the expected temporal locality, avoid cache pollution, and the number of resulting misses in the worst case is predictable. In order to evaluate the FAFBs, we focus on the specific features of our proposal, namely small size and predictability, of paramount importance in hardware for embedded and real-time computing. We do not evaluate its performance in general purpose systems due to results would just reflect the relation between the sizes of the data arrays and the baseline cache. That is, a baseline data cache large enough to store a tile would get little benefits when adding FAFBs, and otherwise the improvements would be those provided by tiling [10]. So, we must compare our results with systems without FAFBs whose analysis is amenable for real-time systems. With such consideration, we must discard conventional data caches, since their WCET analysis is complex and prone to overestimations [2, 3, 22, 23]. Nevertheless, we provide an analytical study of the hits and misses on a conventional fully-associative LRU data cache. This cache provides the best results that any conventional cache could provide, since it does not suffer from line conflicts. Also, our study shows the optimal result that any WCET analysis method could reach on such cache. Regarding random data caches [8], worst-case static analysis cannot be applied on them, so we do not include them in the analysis. Moreover, the probabilistic WCET analysis applied on such caches cannot be applied if they are coupled with our FAFBs proposal. This points the ACDC as a good candidate to pair with FAFBs. Indeed, ACDC fits well as the partner data cache due to its small size and suitability for an easy worst-case static analysis [9]. This coupled ACDC+FAFBs design will be compared with only ACDC as a baseline model, both in terms of worst-case analysis and hardware size in order to show FAFBs capabilities. Scratchpad memories of the same size will also be compared [24]. For simplicity, we assume that the next level in the memory hierarchy is main memory.

ACDC behavior relies on a small preselected list of program counters of memory instructions (stored in the AC part), each of them pointing to a particular cache line (stored in the DC part). Any memory access looks up the ACDC so it may result in a data hit, but only a given preselected memory instruction may replace its associated cache line. That is, any data miss coming from a not preselected instruction bypasses replacement and can not replace DC contents. So, the preselected instructions should be those referencing for the first time *scalar variables with temporal reuse* (i.e. caching a specific data memory line which will be reused by other instructions and will never be evicted), and those referencing *data structures with spatial reuse* (i.e. caching and reusing close data iteratively). In the first case, the ACDC acts as a locked cache or scratchpad. In the second case, the ACDC acts similarly to a conventional cache, but with a "controlled" replacement. So, the ACDC provides the best features of both scratchpads and conventional caches, providing also an easy worst case analysis. At first glance it could be though that, extending the ACDC with several cache lines per DC entry and a replacement policy such as LRU or FIFO, the resulting hardware would be an extended ACDC with the improvements provided by FAFBs. However, such approach would present important limitations. First, it would have a behavior much harder to predict (similar to a conventional cache) and its worst case could not be analyzed by the proposed theory [9]. Moreover, such a design would be complex in hardware (multiple comparisons in parallel in the critical path) and very inefficient, both in predicted hit ratio and energy consumption, since the extended associativity does not improve the cache behavior on common reuse types (temporal reuse on scalar variables, and spatial reuse with stride 1 element).

5. Results

As outlined above, there are many studies on tiling [10, 11, 12], and many algorithms used in embedded and real-time systems are amenable to tiling [17, 18, 19, 20, 21]. However, developing code for specific platforms (and obtaining its WCET bound) is a complex task, and companies do not distribute it. This means that the publicly available real-time benchmarks (e.g. [25, 26]) are too small and may not be representative. Moreover, in such codes the application of tiling is very limited. The exception is the matrix multiplication code, which we analyze below in detail. Afterwards, we perform our analysis on kernel algorithms with clear applications in embedded and real-time systems.

5.1. Detailed evaluation of matrix multiplication

Matrix multiplication is used in fields such as robotics, signal processing, image processing, and adaptive control systems (e.g. Kalman Filter) [17, 21]. For its evaluation, first we perform a reuse analysis on a matrix multiplication algorithm without tiling, and its worst-case locality analysis with the selected baseline (ACDC working alone). A detailed example of WCET analysis of several matrix multiplication codes with the ACDC can be found in [9]. It includes the mathematical details of the reuse theory required for its automated analysis [10]. Next, we apply tiling and perform the same analysis with ACDC+FAFBs. With the obtained results, ACDC and ACDC+FAFBs are compared in terms of worst-case performance and hardware size. Note that applying tiling without FAFBs in a real-time context would not provide any improvement, since the ACDC working alone cannot capture temporal locality on array data structures. On the other hand, any conventional data cache (e.g. LRU) would be hard to analyze by existing WCET analysis methods, forcing them to overestimate [3, 22, 23]. Nevertheless, we also provide an analytical study of hits and misses on a conventional fully-associative LRU cache at the end of this section in order to provide some insight into its actual behavior.

5.1.1. Matrix multiplication without tiling

In this section, we focus on the optimized version of the matrix multiplication algorithm ($N \times N$ row-major ordered matrices, with i, k, j loop nesting and a temporal variable), shown in Figure 1 (a) [11].

Table 1 summarizes the locality properties of data references in the worst case for the baseline option, which only has the ACDC structure. Column "Reference" shows the data memory references in the algorithm. Next, the reuse type of each reference is shown [10]: (self) temporal reuse T, (self) spatial reuse S, and group (temporal or spatial) reuse G. Memory references present temporal reuse when they access the same memory address repeatedly during program execution, whereas they show spatial reuse when they access close memory addresses during program execution. *Self* reuse refers to a single memory instruction involved in the reuse pattern, and *Group* reuse involves several memory instructions. Such information can be obtained by mathematical expressions (not shown) based on the iteration space in loops and the dimensions of the referenced data structure [9, 10]. Temporal reuse is marked only if the reuse distance is short enough to be exploited, i.e. if it does not depend on N. Next column shows which references should have data cache replacement permission in the ACDC. Considering each reference in isolation, the number of misses in the worst case (column "WC-Miss") can be determined depending on the loop bounds. Calculations for obtaining the values in this column are detailed below. Finally, the hardware size required for the analyzed data memory architecture (discussed at the end of this section) is shown below the table. Note that the isolated computation of each reference is safe due to the ACDC behavior, which prevents that a given reference evicts the content cached by another reference. That is, the ACDC prevents cache pollution in the same way that FAFBs do. Such behavior simplifies very much the analysis of hits and misses compared to a conventional cache, where interferences between data accesses must be taken into account for a worst-case analysis.

First, note that the store reference `z[i][j]` is performed after an equivalent reference (load `z[i][j]`), which means that it is classified as group reuse (G) and will find all its accesses already cached by the previous load. That is, it needs no replacement permission in the ACDC and will never miss. Next, all remaining references walk rows sequentially, i.e. they present spatial reuse (S), so they should be granted

Reference	T S G	ACDC	WC-Miss
z[i][j] (load)	– ✓ –	✓	N^3/L
z[i][j] (store)	– – ✓	–	0
x[i][k]	– ✓ –	✓	N^2/L
y[k][j]	– ✓ –	✓	N^3/L
	HW size: 70.5 bytes		

Table 1: Worst-case data locality in matrix multiplication (Figure 1 (a)) with an ACDC having data lines of L elements.

ACDC replacement permission. Since no reference presents temporal reuse with a reuse distance short enough to be exploited (T), FAFBs would not provide any benefit for this algorithm.

Let us now calculate "manually" the number of misses in Table 1. We assume that each cache line in the ACDC stores L data elements. For clarity, we also assume aligned matrices with N being a multiple of the number of elements L in the ACDC line size. In the load reference `z[i][j]`, considering the deepest loop j only, it accesses N times walking a matrix row, but it has replacement permission in the single cache line associated to the load `z[i][j]` in the ACDC (i.e. it acts as a single-line conventional cache for this load instruction), and each cache line contains L elements, so only one of each L accesses will be a cache miss. Hence, the number of misses in loop j is N/L. Since this walk is performed N times in loop k, this means N^2/L. Furthermore, these misses are repeated N times (loop i), so the total number of misses for the load `z[i][j]` is N^3/L. The number of misses for `y[k][j]` can be calculated exactly in the same way and with the same result. Regarding `x[i][k]`, it walks a row sequentially (N/L misses) in loop k, which is repeated N times in loop i, resulting in N^2/L misses.

Given that the number of misses in the worst case is known, either with granted replacement permission (as calculated above) or without it (always miss), in order to perform a WCET optimization, the best references to be granted data cache replacement permission can be obtained [9].

Regarding the hardware size required for this data cache architecture, we assume the parameters described in Section 4. Each ACDC entry requires the storage of the PC, 4 bits for its replacement policy, a particular data cache line, its tag and its state bits [9], which account for 188 bits (23.5 bytes). Thus, the non-tiled version would require 564 bits (70.5 bytes) in the baseline option.

5.1.2. Tiled matrix multiplication

Even considering that the previous results without tiling are good for the small ACDC cache size, probably they could be improved by applying tiling transformations (Figure 1 (b)). The tiling transformation on this algorithm and its theoretical reuse description is detailed in [10]. The deepest loops walk inside the tile, whereas the added outer loops move accross tiles. Such temporal reuse cannot be exploited in the ACDC and, using a conventional cache, a worst case analysis should explore all possible conflicts combined with all possible starting states (contents) in the conventional cache, which is too hard to perform.

FAFBs provide an easily analyzable worst case and at the same time exploit temporal reuse, provided that a tile fits in a FAFB. In this way, once cached, it is reused multiple times without misses until moving to the next tile. Since the FAFB structure is known (B cache lines, with each line storing L consecutive elements), setting the tile size is straightforward for any automatic tiling transformation. In this case, the most adequate tile is one with size of $B \times L$ elements (B rows by L columns), which fits in one FAFB.

Table 2 shows the same information as Table 1 when adding the FAFBs operation. The store `z[i][j]` presents group reuse, with no misses. All remaining references now occur inside a tile so, additionally to the spatial reuse, they may present temporal reuse.

As above, let us account the number of misses. For the load `z[i][j]`, the deepest loop (j) performs L iterations, which means L consecutive accesses. Considering a matrix aligned to memory boundaries (base address of the matrix aligned to cache line size), ACDC replacement permission for this reference, and cache lines of L elements, there is $L/L = 1$ miss in loop j. The next loop (k) iterates B times, but note that `z[i][j]` does not depend on k and all accesses in the deepest loop j are already cached (temporal locality),

Reference	T S G	ACDC	FAFB	WC-Miss
z[i][j] (load)	✓ ✓ –	✓	–	$N^3/(BL)$
z[i][j] (store)	– – ✓	–	–	0
x[i][k]	– ✓ –	✓	–	N^3/L^2
y[k][j]	✓ ✓ –	–	✓	N^2/L
	HW size: 130 bytes			

Table 2: Worst-case data locality in the tiled matrix multiplication (Figure 1 (b)) with ACDC coupled to one FAFB.

so loop k adds no misses. Next, loop i repeats N times the misses in its body. Loop jj repeats N/L times the previous misses. Finally, loop kk repeats N/B times the number of previous misses. This results in $1 \cdot 1 \cdot N \cdot N/L \cdot N/B = N^3/(BL)$ misses for the `z[i][j]` load.

For `y[k][j]`, the deepest loop (j) performs L iterations, which fit in a cache line (1 miss). Loop k performs B iterations, filling the B cache lines of one FAFB (B misses). Loop i does not affect the reference `y[k][j]`, so it repeats the previous accesses, which are already cached (temporal locality), adding no miss. Finally, loop jj repeats N/L times the previous misses and kk repeats N/B times the number of previous misses. This results in $1 \cdot B \cdot 1 \cdot N/L \cdot N/B = N^2/L$ misses for the `y[k][j]` access.

Finally, reference `x[i][k]` iterates B times in loop k and has ACDC replacement permission, i.e. B/L misses. Loop i repeats those misses N times. Next, the previous misses are repeated N/L times in loop jj and N/B times in loop kk. This results in $B/L \cdot N \cdot N/L \cdot N/B = N^3/L^2$.

Regarding the hardware size required for this data memory architecture, we assume again the parameters described in Section 4. Each ACDC entry requires 188 bits (23.5 bytes) [9], whereas each FAFB requires 664 bits (83 bytes). Thus, the tiled version would require 1040 bits (130 bytes) in the ACDC+FAFBs architecture.

5.1.3. Discussion on matrix multiplication

Considering all the misses in the initial non-tiled matrix multiplication algorithm (Figure 1 (a), Table 1, baseline ACDC), it results in a total of $2N^3/L + N^2/L$ misses in the worst case. Table 2, ACDC+FAFBs) is $N^3/BL + N^3/L^2 + N^2/L$. Dividing both expressions, the miss decrement factor for the matrix multiplication algorithm (MDF_{mm}) can be obtained:

$$MDF_{mm} = \frac{\frac{2N^3}{L} + \frac{N^2}{L}}{\frac{N^3}{BL} + \frac{N^3}{L^2} + \frac{N^2}{L}} \approx \frac{2BL}{B+L}$$

That is, the number of misses in the worst case in the tiled matrix multiplication with FAFBs is much lower than in the non-tiled algorithm. Obviously, such factor must be corrected with the additional execution time due to the two additional loops required to perform the tiling, which is the main drawback of tiling transformations. However, it is important to note that the instruction flow has been deeply studied in WCET analysis. Pipelining and branch prediction should provide a potential execution speed of one instruction per cycle [27]. Also, hardware proposals such as instruction cache locking and prefetch (e.g. [28]) and detailed analysis methods on conventional instruction caches (e.g. [2]) may guarantee to always hit in certain chunks of instructions. Furthermore, there are architectures that provide features such as the zero-overhead loop buffer (ZOLB) which may improve loop execution [29]. Hence, assuming that the instruction flow has a fair performance, it is reasonable to consider the data miss delays as the bottleneck, since the instruction fetch and execution latency will be hidden by data miss delays.

Nevertheless, in order to consider all these factors (and not only the number of misses), we have compiled both matrix multiplication algorithms (Figure 1) with GCC-4.4.5 for a 32-bit ARM processor (*arm-linux-gnueabi*). The *min()* function in the tiled version has been set *inline* above the corresponding loop (*loop invariant*). For each algorithm, we have used array sizes (multiple of $L = 4$) of 12×12, 100×100, and

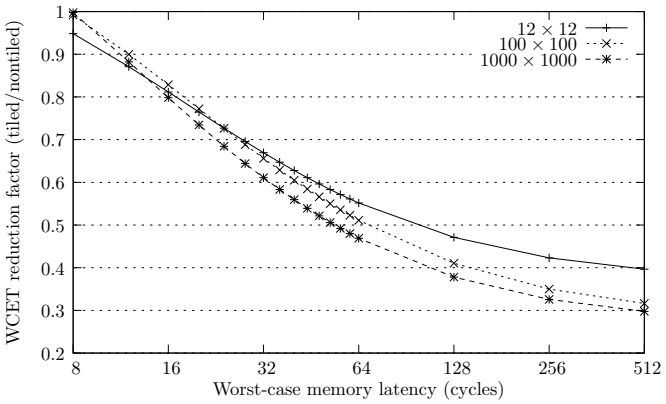


Figure 4: WCET reduction when using FAFBs and tiling on matrix multiplication of $N \times N$ element arrays.

$1\,000 \times 1\,000$ 4-byte integer elements. For each executable file, we have analyzed its WCET assuming $B = 4$ and $L = 4$ for a wide range of memory latencies using Lock-MS [9, 30].

Figure 4 shows the WCET reduction factor between the two cases: WCET bound of the tiled version with FAFBs divided by WCET bound of the non-tiled version. Such factor reflects the reduction in the execution time in the worst case that our proposal would provide. Tiling transformations reduce the number of data memory misses so, as expected, the higher the miss cost (memory latency), the better the tiling results. Also, trends in Figure 4 validate the theoretical results. That is, with $B = 4$ and $L = 4$, $MDF_{mm} \approx 4$, and the WCET reduction factor in the figure tends to $1/MDF_{mm}$ as the memory latency grows.

Depending on the hardware architecture, memory latency may differ very much. On-chip main memories could present latencies around 10 cycles, so in such case our proposal would provide minor benefits. However, studies considering off-chip memories assume latencies such as 38 cycles [6] or 64 cycles [31]. For such systems, our proposal would reduce execution times to one half approximately. It is worth mentioning that, even with low memory latencies, scheduling analysis of multicore real-time systems may force to assume larger memory latencies in the worst case due to possible delays when using a shared memory bus [32]. In such case, the memory latency to consider is proportional to the number of active cores in the processor. For instance, considering a pipelined packed switching bus and memory bank latencies of 4 cycles, the worst-case memory latency would be around 4 times the number of active cores [32]. That is, 8 active cores would mean a worst case on-chip memory latency around 32 cycles. If both off-chip memory and forced delays due to multicore scheduling were to be considered, memory latency in the worst case could also be larger.

Let us now analyze the size required for both (ACDC and ACDC+FAFBs) memory architectures. As calculated above, the non-tiled version would require 70.5 bytes in our analyzed hardware, and the tiled version would require 130 bytes. Let us now assume a scratchpad of the same size. Even more, let us assume that all of such size in the scratchpad stores data, and any tag and state bits are stored apart (similarly as described in [24]). In order to fully store the three matrices in such a scratchpad, each of them should be smaller than 6×6 bytes. Alternatively, storing just one of them (which would provide worse results than our proposal), the matrix could be up to 11×11 bytes, that is, less than a 3×3 integer matrix. This means that such size in a scratchpad is too small to provide significant improvements compared with our proposal.

On the other hand, a lockable cache could be used as a software-controlled prefetch buffer by continuously replacing its contents at run-time [7], but it would imply increasing the number of instructions to execute in order to obtain, by software, precisely the dynamic behavior that our proposal shows autonomously. Moreover, the hardware required for such software-controlled prefetch buffer would require fine grained locking, which means a tag and state bits per data line, as a conventional cache. Indeed, the most straightforward hardware for such behavior seems a fully-associative buffer with a FIFO replacement policy, which is a part of our proposal. Hence, even assuming a similar behavior, such software-controlled buffer would require much more instructions to achieve the same results.

Thus, our approach is straightforward to apply and analyze in a predictable way, and allows us to

estimate a precise bound on the number of misses in the worst case, which enables an accurate and easy WCET computation. Additionally, for systems with off-chip memories or multicore processors, our proposal would reduce significantly the WCET. This is achieved with a very small and simple hardware structure, much smaller than the size required in a scratchpad to obtain similar results.

5.1.4. Tiled matrix multiplication on a conventional LRU cache

In this section we compare our previous results with those of a conventional cache. In general, it is well known that analyzing the WCET in presence of conventional caches is hard [1]. Whereas the WCET analysis with FAFBs accounts the hits and misses separately for each reference, with a conventional cache, accesses coming from all references are interleaved, so they affect each other. Also, data accesses that would cause pollution in conventional caches are avoided in FAFBs by the filtering feature provided by the FPCs (Figure 2). Moreover, on the FAFBs the effects of a given reference are always the same, whereas on set-associative or direct-mapped caches, a given load/store executed several times may affect different sets, so its effects on the cached contents may present critical differences. In order to analyze the WCET considering all these problems, most methods provide an overestimated result (e.g. [3, 22, 23]). However, since we focus on a small particular algorithm (Figure 1 (b)), hits and misses can be accounted analytically for a fully-associative LRU data cache without overestimation. A fully-associative cache is the conventional cache that provides the best WCET results, since set-associative caches may present line conflicts. Also, the LRU replacement policy provides the best results in terms of temporal reuse. Finally, note that such hits and misses represent the optimal result that any WCET analysis method could reach, since our analytical study provides exact hit and miss values.

We consider a maximum cache size of less than N cache lines, since N depends on the problem and tiling is applied precisely to avoid caching whole structures (whole rows or columns in this case). As discussed previously, the store `z[i][j]` presents group reuse (G) (Table 2), so it always hits. Next, reference `x[i][k]` presents spatial reuse only (S), i.e. it never repeats accesses, so a single cache line is required to exploit the sequential accesses. That is, any conventional cache ensuring that this single line is not evicted will perform exactly as in Table 2, requiring at least 1 cache line (as the one associated to the ACDC in this table). For the load reference `z[i][j]`, in order to exploit its spatial reuse, it must cache the data accessed in the loop j. This loop performs L iterations, so it would need a single cache line (storing these L elements). This reference presents temporal reuse in loop k, since its accessed data do not depend on k. Thus, as long as the previous single cache line is not evicted in loop k, the temporal reuse of this reference can be fully exploited. Again, this means that the number of misses for this reference coincides with that of Table 2 and that a conventional cache would need at least 1 cache line (as the one associated to the ACDC in this table). Finally, reference `y[k][j]` also presents temporal and spatial reuse. As above, spatial reuse is present in loop j, requiring a single cache line to exploit it. In loop k (B iterations), B cache lines are required to cache all its data accesses. Such data are reused in loop i, since accesses of `y[k][j]` do not change when i varies. Thus, the temporal reuse of this reference is fully exploited as long as the previous B cache lines are not evicted. Once again, this means that the number of misses for this reference coincides with that of Table 2 and that a conventional cache would need at least B cache lines (as the FAFB in this table).

Summarizing, the performance in the worst case of the tiled version of the matrix multiplication algorithm in presence of any conventional LRU cache will be equal than our proposed ACDC+FAFBs, assuming that it has sufficient cache lines. Also, the number of cache lines required for such conventional cache would be equal (fully-associative) or higher (set-associative, which may present line conflicts) than our proposed solution, and additional cache lines would not achieve better results.

Although this analysis is not detailed for the kernel algorithms below, all of them show the same behavior (the same performance, requiring the same number of cache lines) but the *unbalanced footprints*, where a conventional cache would require more cache lines in order to reach the results achieved by ACDC+FAFBs. This case is discussed below. Thus, considering that in general analyzing the WCET in presence of conventional caches is harder, and that, even without overestimations, conventional caches cannot improve our results, our proposal seems a very promising option.

```
for (i=0; i<N; i++) {              // N iterations
  r=c[i];
  for (j=0; j<M; j++)              // M iterations
    a[i][j]=b[j]*r;
}
```

Figure 5: Algorithm with unbalanced footprints [12].

```
for (jj=0; jj<M; jj+=BL)                  // M/BL it
  for (i=0; i<N; i++) {                     // N it
    r=c[i];
    for (j=jj; j<min(jj+BL,M); j++)         // BL it
      a[i][j]=b[j]*r;
}
```

Figure 6: Tiling transformation on unbalanced footprints (Figure 5) [12].

5.2. Evaluation of kernel algorithms

In this section we repeat the previous analysis in a set of algorithms widely used in many areas, including real-time systems. For each algorithm, we analyze each data reference separately, and calculate its cache misses in the worst case. That is, there is no need of analyzing the evictions that a data reference may cause to the cached data of other references (which would occur in a conventional data cache) because both ACDC and FAFBs prevent them. Since the calculation of misses considers only one reference at a time, it is easy to perform and can be automated based on the reuse theory [10]. With the analysis of each data reference, an accurate bound on its number of misses in the worst case can be obtained. Then, adding the misses of each reference, a bound on the number of misses of the whole algorithm is obtained.

In order to compare the tiled version of each algorithm, predictable through ACDC+FAFBs, with its non-tiled version, suitable without FAFBs, we compute the miss decrement factor (MDF = # misses Non-Tiled / # misses Tiled). For clarity, in this section we do not show the references with group reuse that add no cache misses. Finally, we also provide the size required by the analyzed cache structures, including both control and data bits. Such size should provide an intuitive idea of the efficiency achieved, which may be useful to compare with the results that a scratchpad of the same size could achieve.

Unbalanced footprints. In these algorithms, data structures of different sizes are traversed. Figure 5 shows a simple algorithm with unbalanced footprints, and Figure 6 shows its tiled version, as they appear as *Programs 18a/b* in [12]. Table 3 summarizes the locality information on them. In the tiled version, since the temporal reuse is found through sequential accesses in reference `b[j]`, it should be granted FAFB replacement permission and the tile size should be $1 \times BL$ elements. References `a[i][j]` and `c[i]` present spatial reuse only, so they would be granted replacement permission in ACDC. The miss decrement factor in the unbalanced footprints algorithm is:

$$MDF_{uf} = \frac{\frac{NM}{L} + \frac{NM}{L} + \frac{N}{L}}{\frac{NM}{L} + \frac{M}{L} + \frac{NM}{BL^2}} \approx \frac{2BL}{BL+1} \approx 2$$

As outlined in Section 5.1.4, the tiled version of this algorithm gets the same performance using an ACDC+FAFBs than using a conventional fully-associative LRU data cache. However, in order to get such performance, the fully-associative cache must have more cache lines than those required for the ACDC+FAFBs. This is due to the pollution generated by reference `a[i][j]`, i.e. data that are cached unnecessarily and may evict reusable data. In this algorithm, `b[j]` exploits its temporal reuse as long as the $B \times L$ data accessed in loop j are not evicted. This requires B cache lines (of L elements). However, interleaved with these data accesses, reference `a[i][j]` also performs $B \times L$ accesses. These accesses do not

	Non-tiled (ACDC)			Tiled (ACDC+FAFBs)			
Reference	T S G	ACDC	WC-Miss	T S G	ACDC	FAFB	WC-Miss
a[i][j]	– ✓ –	✓	$\frac{NM}{L}$	– ✓ –	✓	–	$\frac{NM}{L}$
b[j]	– ✓ –	✓	$\frac{NM}{L}$	✓ ✓ –	–	✓	$\frac{M}{L}$
c[i]	– ✓ –	✓	$\frac{N}{L}$	– ✓ –	✓	–	$\frac{NM}{BL^2}$
	HW size: 70.5 bytes			HW size: 130 bytes			

Table 3: Worst-case data locality in unbalanced footprints (Figures 5, 6).

```
for (i=1; i<N-1; i++)                    // N-2 it
  for (j=1; j<N-1; j++)                  // N-2 it
    a[i][j]=0.25*(a[i+1][j]+a[i-1][j]+
                  a[i][j+1]+a[i][j-1]);
```

Figure 7: Stencil computation algorithm.

present temporal reuse, so caching them does not provide any benefit. However, to ensure that these accesses do not evict those from reference `b[j]` (which will be reused) a fully-associative conventional cache needs at least $2B$ cache lines. Our proposal does not suffer from this pollution, since only references with temporal reuse (`b[j]` in this case) have replacement permission on the FAFB. So, pollution is avoided and reference `a[i][j]` uses just the single cache line (in the ACDC) that needs to exploit its spatial reuse. Therefore, although a conventional fully-associative LRU data cache would perform as our ACDC+FAFBs, it would require $2B + 1$ cache lines, whereas the ACDC+FAFBs would use $B + 2$ cache lines.

Stencil algorithms. In these codes, elements in a multidimensional data structure are processed following a fixed pattern, so that each result depends on its surrounding elements. Such codes are used, for instance, in image processing, the Jacobi kernel, and the Gauss-Seidel method. Figure 7 shows an example of such algorithms. Although FAFBs could be fully exploited when applying tiling transformations on them, accounting the number of misses is not straightforward in this case. Thus, since our main motivation is to highlight the simplicity that the FAFBs provide for exploiting the temporal reuse (and in turn its WCET analysis), we discard this option despite its good results. Nevertheless, stencil codes are a good example to show that FAFBs can reduce the number of misses even without compiler optimizations, as long as the temporal reuse distance in the code fits in the FAFBs. Table 4 shows the reuse information when using and not using FAFBs on the algorithm in Figure 7. The miss decrement factor is:

$$MDF_{sc} = 5/3$$

		ACDC		ACDC+FAFBs		
Reference	T S G	ACDC	WC-Miss	ACDC	FAFB	WC-Miss
a[i+1][j]	– ✓ ✓	✓	$(N-2)^2/L$	✓	–	$(N-2)^2/L$
a[i-1][j]	– ✓ ✓	✓	$(N-2)^2/L$	✓	–	$(N-2)^2/L$
a[i][j+1]	– ✓ ✓	✓	$(N-2)^2/L$	–	✓	$(N-2)^2/L$
a[i][j]	– ✓ ✓	✓	$(N-2)^2/L$	–	–	0
a[i][j-1]	– ✓ ✓	✓	$(N-2)^2/L$	–	–	0
	HW size: 117.5 bytes			HW size: 130 bytes		

Table 4: Worst-case data locality in stencil computation (Figure 7).

```
for (k=0; k<M; k++){  //For all columns; M it
  /******** begin non-analyzed area ********/
  imax=findPivotColumn(a,k);
  if (a[imax][k]==0) return; //Singular matrx
  swapRows(a,k,imax);
  /********* end non-analyzed area *********/
  tkk=a[k][k];                    // Make a[k,k]=1
  for (j=k+1; j<N; j++)            // N-(M+1)/2 it
    a[k][j]=a[k][j]/tkk;           /* a[k][j]_1 */
  a[k][k]=1.0;
  for (i=0;i<M;i++) //Make identity col; M it
    if (i!=k){
      tik=a[i][k];
      for (j=k+1; j<N ;j++)    // N-(M+1)/2 it
        a[i][j]=a[i][j]-a[k][j]*tik;
      a[i][k]=0;       /* a[k][j]_2 */
    }
}
```

Figure 8: Matrix inversion algorithm (Gauss-Jordan elimination).

Matrix inversion. Matrix inversion is commonly used in real-time applications such as multiple-input multiple-output (MIMO) communications systems such as 802.11n, 3D graphics rendering, 3D simulations, cryptographic algorithms, and adaptive control systems [18, 19, 21]. Figure 8 shows the matrix inversion algorithm, specifying the area skipped in the analysis, and Figure 9 shows its tiled version. Table 5 shows the locality summary. It can be seen that the tiling transformation with a FAFB achieves the same number of misses in `a[i][j]`, but reduces those of `a[k][j]`. The miss decrement factor in the analyzed area of the matrix inversion algorithm is:

$$MDF_{mi} \approx \frac{2B}{B+1}$$

LU decomposition. This algorithm is used, for instance, in real-time adaptive control systems (e.g. Kalman Filter) [21]. It rewrites a matrix A as the product of two triangular matrices L (lower triangular) and U (upper triangular), which are stored into a single matrix structure (usually overwriting A) combining L and U. Having these matrices, many operations are much faster on them than on the original matrix. In real-time contexts, this algorithm is used for instance in computer vision systems [20]. Figure 10 shows the LU decomposition algorithm, and Figure 11 shows its tiled version [10]. Table 6 shows a summary of its locality. In this case, without FAFBs, the number of misses is not straightforward to calculate. Indeed, Table 6 contains two approximations (expressions $L-1$ and $L-1.5$) to bound the worst case. That is, the inherent predictability of the FAFBs, apart from improving results, allows a more precise estimation of the number of misses. Using our hardware proposal, a rather roughly approximation (see specific values in Table 7 below) of the miss decrement factor in LU decomposition is:

$$MDF_{lu} \sim 2L$$

As a general summary, all the studied algorithms require only one FAFB but one, which requires two FAFBs. Such small hardware can lead to significant and predictable performance improvements. Thus, adding a few FAFBs to the memory hierarchy of embedded and real-time systems seems advisable.

5.3. Discussion on parameters

Based on the previous analytical results, in this section we provide the numerical values that would be obtained for specific scenarios and discuss the results. We consider FAFBs with 2 to 8 cache lines (B), with

```
for (k=0; k<M; k++){   //For all columns; M it
  [...]                                  // omitted code
  tkk=a[k][k];
  a[k][k]=1.0;
  for (ii=0; ii<M; ii+=B){                 // M/B it
    for (wi=0;wi<min(ii+B,M)-ii;wi++)   //B it
      if (ii+wi!=k)
        vtik[wi]=a[ii+wi][k];
    for (j=k+1; j<N; j++){      // N-(M+1)/2 it
      if (ii==0)
        a[k][j]=a[k][j]/tkk;
      tkj=a[k][j];
      for (i=ii; i<min(ii+B,M); i++){   //B it
        if (i!=k)
          a[i][j]=a[i][j]-tkj*vtik[i-ii];
      }
    }
  }
}
```

Figure 9: Tiling transformation on the matrix inv. algorithm (Figure 8).

	Non-tiled (ACDC)			Tiled (ACDC+FAFBs)			
Ref.	T S G	ACDC	WC-Miss	T S G	ACDC	FAFB	WC-Miss
a[k][j]$_1$	– ✓ –	✓	$\left(N-\frac{M+1}{2}\right)\frac{M}{L}$	– ✓ ✓	✓	–	$\left(N-\frac{M+1}{2}\right)\frac{M^2}{BL}$
a[k][j]$_2$	– ✓ –	✓	$\left(N-\frac{M+1}{2}\right)\frac{M^2}{L}$				
a[k][k]	– – –	–	M	– – –	–	–	M
a[i][k]	– – –	–	M^2-M	– – –	–	–	M^2-M
vtik[wi]				✓ ✓ ✓	✓	–	1
a[i][j]	– ✓ –	✓	$\left(N-\frac{M+1}{2}\right)\frac{M^2}{L}$	✓ ✓ –	–	✓	$\left(N-\frac{M+1}{2}\right)\frac{M^2}{L}$
	HW size: 70.5 bytes			HW size: 130 bytes			

Table 5: Worst-case data locality in matrix inversion (Figures 8, 9).

```
for (i=0; i<N; i++) {
  r = a[i][i];
  for (j=i+1; j<N; j++) {
    a[j][i]=a[j][i]/r;
    s=a[j][i];
    for (k=i+1; k<N; k++)
      a[j][k]=a[j][k]-s*a[i][k];
  }
}
```

Figure 10: LU decomposition algorithm [10].

```
for (jj=0; jj<N; jj+=B)
 for (kk=0; kk<N; kk+=L)
  for (i=0; i<n; i++) {
   if ((kk<=(i+1)) && ((i+1)<(kk+L)))
     r = a[i][i];
   for (j=max(i+1,jj); j<min(jj+B,N); j++){
    if ((kk<=(i+1)) && ((i+1)<(kk+L))) {
      a[j][i]=a[j][i]/r;
      s=a[j][i];
    }
    for (k=max(i+1,kk); k<min(kk+L,N); k++)
      a[j][k]=a[j][k]-s*a[i][k];
   }
  }
```

Figure 11: Tiling transformation on LU decomposition (Figure 10) [10].

	Non-tiled (ACDC)			Tiled (ACDC+FAFBs)			
Reference	T S G	ACDC	WC-Miss	T S G	ACDC	FAFB	WC-Miss
a[i][i]	– – –	–	N	– – –	–	–	N
a[j][i]	– – –	–	$\frac{N^2-N}{2}$	✓✓ –	–	✓	$\frac{N^2}{2L} - \frac{N}{L}$
a[i][k]	– ✓ –	✓	$\left(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6}\right) \cdot \frac{1}{L-1}$	✓✓ –	✓	–	$\left(\frac{\left(\frac{N}{L}\right)^3}{3} + \frac{\left(\frac{N}{L}\right)^2}{2} + \frac{\frac{N}{L}}{6}\right) L - \left(\frac{N}{L}\right)^2$
a[j][k]	– ✓ –	✓	$\left(\frac{N^3}{3} + \frac{N^2}{2} + \frac{N}{6}\right) \cdot \frac{1}{L-1.5}$	✓✓ –	–	✓	$\frac{(N-1)^2}{L}$
	HW size: 47 bytes			HW size: 189.5 bytes			

Table 6: Worst-case data locality estimation in LU decomposition (Figures 10, 11).

each line containing 2 to 8 elements (L). E.g. if we consider that these elements are 32-bit integer numbers, this means lines of 16 bytes for $L = 4$. Also, we consider a problem size with N ranging from 10 to 10 000, always with $M = N/2$. Table 7 shows the obtained miss decrement factors.

Varying the size of the problem N is significant for the LU decomposition and to some extent for matrix multiplication, but not so much for the other algorithms. This means that the worst-case performance of our proposal is independent of the problem size, contrarily to conventional caches. Moreover, note that the larger the problem size, the higher the miss decrement factor.

When varying the size of the line L, both matrix multiplication and LU decomposition are significantly affected.

Finally, the miss decrement factor when varying the number of cache lines B in the FAFB affects matrix multiplication significantly, and also matrix inversion in a minor degree.

These numerical results follow the simplified analytical expressions obtained above. As expected, the larger the FAFBs, the better the results. This is not always true with conventional caches, due to they are affected by collisions and pollution. Also, contrarily to conventional caches, the larger the data structures in tasks, the higher the miss decrement factor in the worst case.

From a hardware perspective, varying such values would affect performance as follows. Increasing the number of FAFBs would allow the speed-up of algorithms that, when tiling is applied, present more than two sequences of accesses on non-scalar variables with temporal locality at the same time. In the kernel algorithms studied, no more than 2 FAFBs are required. If several of these algorithms were used in the same task but at different times (phases) of the execution, the instruction with FAFB replacement permission could be changed at run-time for the different phases of the task. The drawback of increasing the number of FAFBs is the possible delay due to increasing the "caching size" and its corresponding energy consumption due to the parallel comparisons of the tags. Nevertheless, current general purpose processors such as ARM600 and POWER3-II have associativities of 64 and 128-way, which is far beyond our current proposal. Also, energy wasting could be avoided by disabling each FAFB when it is not used, which is known at compile-time. Regarding the number of entries in each FAFB (B), their increment would allow to cache more data, specially for large strides. The drawbacks would be the same as increasing the number of FAFBs. Finally, increasing the size of the cache lines (L) would benefit the temporal locality for very small strides, e.g. for sequential accesses. Increasing this size would have minor drawbacks, since the number of tags to compare in look up does not change, although in cases of large strides it would cache unnecessary data. This parameter is closely related to the data bus width.

6. Conclusions

Calculating an accurate bound on the WCET of a task is required in real-time systems. This is specially difficult for tasks that stress the data cache. In this paper we propose a new hardware structure (FAFB) for data caching to work in conjunction with a coupled data cache (e.g. ACDC in real-time systems). This new structure is able to take profit of temporal reuse on array data structures in a predictable way, which enables a precise and relatively easy WCET analysis.

The main features of the FAFB are the following: small size (no more than 2 FAFBs, each of them with 4 cache lines by default, are needed in our tested kernels), prevents pollution (accesses of a reference do not evict data cached by other references), energetically efficient (FAFBs could be enabled/disabled on request at run-time), specially suitable for real-time systems due to its predictable behavior, and specially suitable for embedded devices due to its high efficiency and small size.

Since our proposal exploits temporal reuse on array data structures, it works specially well on tiled algorithms. Tiling transformations are well known optimization techniques to reduce the temporal reuse distance. With our proposed architecture, setting the tile size of tiling transformations is straightforward given the number of entries and line size of the implemented FAFBs. Regarding temporal reuse, our proposal achieves the best performance that any conventional cache could reach, requires the same or less cache lines to achieve such performance, and presents a completely predictable behavior.

Our results show that the number of misses in the worst case can be reduced between 1.3 and 19 times on the analyzed kernels in respect of its non-tiled version on an ACDC working alone. For real-time systems

Matrix multiplication	$L=2$			$L=4$			$L=8$		
	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$
$N=10$	1.91	2.47	2.90	2.47	3.50	4.42	2.90	4.42	6.00
$N=100$	1.99	2.64	3.17	2.64	3.94	5.22	3.17	5.22	7.73
$N=1000$	2.00	2.66	3.20	2.66	3.99	5.32	3.20	5.32	7.97
$N=10000$	2.00	2.67	3.20	2.67	4.00	5.33	3.20	5.33	8.00
Unbalanced footprints	$L=2$			$L=4$			$L=8$		
	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$
$N=10$	1.63	1.80	1.89	1.80	1.89	1.94	1.89	1.94	1.97
$N=100$	1.60	1.78	1.88	1.78	1.88	1.94	1.88	1.94	1.97
$N=1000$	1.60	1.78	1.88	1.78	1.88	1.94	1.88	1.94	1.97
$N=10000$	1.60	1.78	1.88	1.78	1.88	1.94	1.88	1.94	1.97
Stencil computation	$L=2$			$L=4$			$L=8$		
	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$
$N=10$	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67
$N=100$	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67
$N=1000$	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67
$N=10000$	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67	1.67
Matrix inversion	$L=2$			$L=4$			$L=8$		
	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$
$N=10$	1.39	1.62	1.76	1.34	1.52	1.63	1.26	1.40	1.47
$N=100$	1.34	1.60	1.78	1.33	1.59	1.76	1.32	1.57	1.73
$N=1000$	1.33	1.60	1.78	1.33	1.60	1.78	1.33	1.60	1.77
$N=10000$	1.33	1.60	1.78	1.33	1.60	1.78	1.33	1.60	1.78
LU decomposition	$L=2$			$L=4$			$L=8$		
	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$	$B=2$	$B=4$	$B=8$
$N=10$	6.27	6.27	6.27	4.22	4.22	4.22	2.60	2.60	2.60
$N=100$	10.92	10.92	10.92	9.51	9.51	9.51	13.67	13.67	13.67
$N=1000$	11.88	11.88	11.88	11.46	11.46	11.46	18.08	18.08	18.08
$N=10000$	11.99	11.99	11.99	11.71	11.71	11.71	18.89	18.89	18.89

Table 7: Decrement factor in the number of misses in the worst case when varying the number of lines B in each FAFB, the size of each line L and the problem size N (with $M = N/2$).

with off-chip memories or multicore processors, such results may imply a significant reduction in the WCET. This is achieved with a hardware structure much smaller than the size required in a scratchpad to obtain similar results.

[1] L. C. Aparicio, J. Segarra, C. Rodríguez, J. L. Villarroel, V. Viñals, Avoiding the WCET overestimation on LRU instruction cache, in: Proc. 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 08), IEEE Computer Society Press, Kaohsiung, Taiwan, 2008, pp. 393–398. doi:10.1109/RTCSA.2008.10.

[2] Y. T. S. Li, S. Malik, A. Wolfe, Cache modeling for real-time software: beyond direct mapped instruction caches, in: Proc. of the IEEE Real-Time Systems Symposium, 1996, pp. 254–264. doi:10.1109/REAL.1996.563722.

[3] R. White, F. Mueller, C. Healy, D. Whalley, M. Harmon, Timing analysis for data caches and set-associative caches, in: Proc. of the IEEE Real-Time Technology and Applications Symposium, 1997, pp. 192–202. doi:10.1109/RTTAS.1997.601358.

[4] V. Suhendra, T. Mitra, A. Roychoudhury, T. Chen, WCET centric data allocation to scratchpad mem-

ory, in: Proceedings of the 26th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, 2005, pp. 223–232. doi:10.1109/RTSS.2005.45.

[5] J.-F. Deverge, I. Puaut, WCET-directed dynamic scratchpad memory allocation of data, Euromicro Conference on Real-Time Systems (2007) 179–190doi:10.1109/ECRTS.2007.37.

[6] X. Vera, B. Lisper, J. Xue, Data caches in multitasking hard real-time systems, in: Proc. of the IEEE Real-Time Systems Symp., 2003, pp. 154–166. doi:10.1109/REAL.2003.1253263.

[7] J. Cong, H. Huang, C. Liu, Y. Zou, A reuse-aware prefetching scheme for scratchpad memory, in: Proceedings of the 48th Design Automation Conference, DAC '11, ACM, New York, NY, USA, 2011, pp. 960–965. doi:10.1145/2024724.2024937.

[8] F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, D. Maxim, Proartis: Probabilistically analyzable real-time systems, ACM Trans. Embed. Comput. Syst. 12 (2s) (2013) 94:1–94:26. doi:10.1145/2465787.2465796.

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

[10] M. E. Wolf, M. S. Lam, A data locality optimizing algorithm, SIGPLAN Not. 26 (1991) 30–44.

[11] M. D. Lam, E. E. Rothberg, M. E. Wolf, The cache performance and optimizations of blocked algorithms, SIGPLAN Not. 26 (4) (1991) 63–74. doi:10.1145/106973.106981.

[12] M. Wolfe, More iteration space tiling, in: Proceedings of the 1989 ACM/IEEE conference on Supercomputing, Supercomputing '89, ACM, New York, NY, USA, 1989, pp. 655–664. doi:10.1145/76263.76337.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, SIGPLAN Not. 43 (6) (2008) 101–113. doi:10.1145/1379022.1375595.

[14] N. Ahmed, N. Mateev, K. Pingali, Tiling imperfectly-nested loop nests, in: In Proc. of SC 2000, 2000, p. 31. doi:10.1109/SC.2000.10018.

[15] A. Hartono, M. Manik, A. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, Primetile: A parametric multi-level tiler for imperfect loop nests, Tech. Rep. OSU-CISRC-2/09-TR04, Computer Science and Engineering Department, The Ohio State University (2009).

[16] Y. Song, Z. Li, A compiler framework for tiling imperfectly-nested loops, in: L. Carter, J. Ferrante (Eds.), Languages and Compilers for Parallel Computing, Vol. 1863 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 185–200. doi:10.1007/3-540-44905-1_12.

[17] F. Bensaali, A. Amira, A. Bouridane, Accelerating matrix product on reconfigurable hardware for image processing applications, Circuits, Devices and Systems, IEE Proceedings - 152 (3) (2005) 236–246. doi:10.1049/ip-cds:20040838.

[18] L. Ma, K. Dickson, J. McAllister, J. McCanny, Qr decomposition-based matrix inversion for high performance embedded mimo receivers, Signal Processing, IEEE Transactions on 59 (4) (2011) 1858–1867. doi:10.1109/TSP.2011.2105485.

[19] A. Rupp, T. Eisenbarth, A. Bogdanov, O. Grieb, Hardware SLE solvers: Efficient building blocks for cryptographic and cryptanalyticapplications, Integration, the VLSI Journal 44 (4) (2011) 290–304. doi:10.1016/j.vlsi.2010.09.001.

[20] E. H. A. T. Targhi, M. Bjorkman, J.-O. Eklundh, Real-time texture detection using the LU-transform, in: Workshop of Computation Intensive Methods for Computer Vision, 2006.

[21] F. Khan, R. Ashraf, Q. Abbasi, A. Nasir, Resource efficient parallel architectures for linear matrix algebra in real time adaptive control algorithms on reconfigurable logic, in: Second International Conference on Electrical Engineering, ICEE, 2008, pp. 1–9. doi:10.1109/ICEE.2008.4553939.

[22] H. Ramaprasad, F. Mueller, Bounding worst-case data cache behavior by analytically deriving cache reference patterns, in: Real-Time and Embedded Technology and Applications Symposium, 2005, pp. 148–157. doi:10.1109/RTAS.2005.12.

[23] R. Sen, Y. N. Srikant, WCET estimation for executables in the presence of data caches, in: Proceedings of the 7th ACM & IEEE International Conference on Embedded Software, ACM, New York, NY, USA, 2007, pp. 203–212. doi:10.1145/1289927.1289960.

[24] S. Steinke, L. Wehmeyer, B.-S. Lee, P. Marwedel, Assigning program and data objects to scratchpad for energy reduction, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2002, pp. 409–415. doi:10.1109/DATE.2002.998306.

[25] Seoul National University Real-Time Research Group, SNU-RT benchmark suite for worst case timing analysis (2008).

[26] J. Gustafsson, A. Betts, A. Ermedahl, B. Lisper, The Mälardalen WCET benchmarks – past, present and future, in: B. Lisper (Ed.), WCET2010, Brussels, Belgium, 2010, pp. 137–147. doi:10.4230/OASIcs.WCET.2010.136.

[27] R. D. Reutemann, Worst-case execution time analysis for dynamic branch predictors, Ph.D. thesis, University of York (Jan. 2008).

[28] L. C. Aparicio, J. Segarra, C. Rodríguez, V. Viñals, Combining prefetch with instruction cache locking in multitasking real-time systems, in: Proceedings of the IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society Press, Macau SAR, China, 2010, pp. 319–328. doi:10.1109/RTCSA.2010.8.

[29] G.-R. Uh, Y. Wang, D. Whalley, S. Jinturkar, C. Burns, V. Cao, Effective exploitation of a zero overhead loop buffer, SIGPLAN Not. 34 (7) (1999) 10–19. doi:10.1145/315253.314419.

[30] L. C. Aparicio, J. Segarra, C. Rodríguez, V. Viñals, Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems, Journal of Systems Architecture 57 (7) (2011) 695–706. doi:http://dx.doi.org/10.1016/j.sysarc.2010.08.008.

[31] R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, F. Tirado, Stack filter: Reducing L1 data cache power consumption, Journal of Systems Architecture 56 (12) (2010) 685 – 695. doi:DOI: 10.1016/j.sysarc.2010.10.002.

[32] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, M. Valero, Hardware support for wcet analysis of hard real-time multicore systems, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09, ACM, New York, NY, USA, 2009, pp. 57–68. doi:10.1145/1555754.1555764.