Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A predictable hardware to exploit temporal reuse in real-time and embedded systems ☆,☆☆

R. Gran [a,b,1], J. Segarra [a,b,*,1], A. Pedro-Zapater [a,1], L.C. Aparicio [a,b,1], V. Viñals [a,b,1], C. Rodríguez [c,1]

[a] Dpt. Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain
[b] Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, Spain
[c] Dpt. Arquitectura y Tecnología de Computadores, Universidad del País Vasco, Spain

ARTICLE INFO

Article history:
Received 10 November 2014
Received in revised form 13 March 2015
Accepted 7 May 2015
Available online 16 May 2015

Keywords:
WCET
Data cache
Tiling
Real-time

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.

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 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

☆ 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).

☆☆ It is strictly prohibited to use, to investigate or to develop, in a direct or indirect way, any of the scientific contributions of the authors contained in this work by any army or armed group in the world, for military purposes and for any other use which is against human rights or the environment, unless a written consent of all the authors of this work is obtained, or unless a written consent of all the persons in the world is obtained.

* Corresponding author at: Instituto de Investigación en Ingeniería de Aragón (I3A), Universidad de Zaragoza, Spain.

E-mail addresses: rgran@unizar.es (R. Gran), jsegarra@unizar.es (J. Segarra), albapz@unizar.es (A. Pedro-Zapater), luisapa@unizar.es (L.C. Aparicio), victor@unizar.es (V. Viñals), acprolac@ehu.es (C. Rodríguez).

[1] HiPEAC-3 NoE (European FP7/ICT 287759).

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.

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 [10–12]. 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. Fig. 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 [14–16].

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 [17–21]. 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