Journal of Systems Architecture 57 (2011) 695–706

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems ☆

Luis C. Aparicio [a,d], Juan Segarra [a,c,d,*], Clemente Rodríguez [b,d], Víctor Viñals [a,c,d]

[a] DIIS, Universidad de Zaragoza, 50018 Zaragoza, Spain
[b] DATC, Universidad del País Vasco, 20018 San Sebastián, Spain
[c] Instituto de Investigación en Ingeniería de Aragón (I3A), 50018 Zaragoza, Spain
[d] European Network of Excellence on High Performance and Embedded, Architecture and Compilation (HiPEAC)[1]

ARTICLE INFO

Article history:
Received 8 January 2010
Received in revised form 24 June 2010
Accepted 23 August 2010
Available online 31 August 2010

Keywords:
WCET
Instruction cache-locking
Line-buffer

ABSTRACT

In multitasking real-time systems it is required to compute the WCET of each task and also the effects of interferences between tasks in the worst case. This is very complex with variable latency hardware, such as instruction cache memories, or, to a lesser extent, the line buffers usually found in the fetch path of commercial processors. Some methods disable cache replacement so that it is easier to model the cache behavior. The difficulty in these cache-locking methods lies in obtaining a good selection of the memory lines to be locked into cache. In this paper, we propose an ILP-based method to select the best lines to be loaded and locked into the instruction cache at each context switch (dynamic locking), taking into account both intra-task and inter-task interferences, and we compare it with static locking. Our results show that, without cache, the spatial locality captured by a line buffer doubles the performance of the processor. When adding a lockable instruction cache, dynamic locking systems are schedulable with a cache size between 12.5% and 50% of the cache size required by static locking. Additionally, the computation time of our analysis method is not dependent on the number of possible paths in the task. This allows us to analyze large codes in a relatively short time (100 KB with 10^{65} paths in less than 3 min).

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 computed 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. This is mainly due to speculation (control or data) or to the use of hardware components with variable latency. Branch predictors fall in the first category, whereas memory hierarchy and datapath pipelining belong to the second one. 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 it is required to know the previous accesses sequence in order to compute the latency of a given access in advance. Resolving these *intra-task* interferences is a difficult problem its own. Anyway, 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 WCET of an isolated task may change when considering interferences. Cache-locking tackles the whole problem by disabling the cache replacement, so the cache content does not vary. Specifically, for an instruction cache, the instruction fetch hits and misses depend on whether each instruction belongs to a cached and locked memory line and not on the previous accesses.

In this paper, we focus on the instruction fetch path and analyze several configurations of the memory architecture shown in Fig. 1. It consists of a line buffer (LB) and a lockable instruction cache (i-cache). The i-cache retains the fixed subset of instruction lines previously loaded by system software at task switches. It does not need fine-grained locking, but whole cache-locking. The LB has the size of a cache line and acts as a single-line cache memory regarding the tag, access latency and refill latency. The only difference with a conventional cache is that it prevents exploiting any

☆ This work was supported in part by grants TIN2007-66423 (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). This work is an extension of the technical report RR-09-01 with new contributions and experimental results [1].
* Corresponding author at: DIIS, Universidad de Zaragoza, 50018 Zaragoza, Spain.
E-mail addresses: luisapa@unizar.es (L.C. Aparicio), jsegarra@unizar.es (J. Segarra), clemente.rodriguezl@ehu.es (C. Rodríguez), victor@unizar.es (V. Viñals).
[1] http://www.hipeac.net/

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.08.008

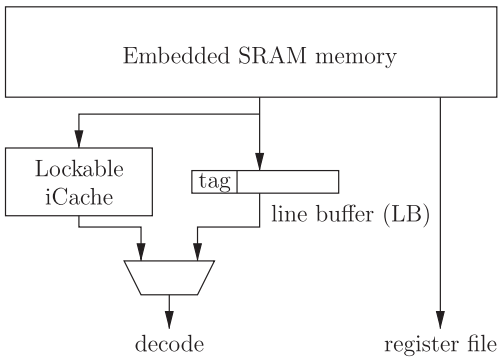

Fig. 1. Memory architecture considered.

temporal locality by invalidating its content on both i-cache hits and backward jumps to the currently buffered line. Instruction cache-locking can be implemented in several ways. One way is to design and synthesize a specific organization taking advantage of its control simplicity, since no replacement algorithm needs to be implemented and the refill happens always in bursts. Another way is making use of the locking capabilities present in many commercial processors devoted to the medium and high-end embedded market.[2]

We propose a new method intended to minimize the instruction cache contribution on the WCET. Previous works studying cache-locking behavior do not distinguish between spatial and temporal locality, so it is not clear how either of these affects the WCET. In order to evaluate the importance of spatial locality we first analyze an instruction line buffer (LB) working alone. Second, to analyze the impact on the WCET of exploiting temporal locality, we add an instruction cache, managed under static and dynamic locking, to the LB. In *static locking*, memory lines are preloaded at system start-up and remain unchanged during the whole system lifetime. We use the *Minimize Utilization* (Lock-MU) method for selecting the memory lines to be locked into the instruction cache [2]. In *dynamic locking*, the instruction cache is preloaded and locked in each context switch, so that there is one selection of memory lines per task. To obtain this selection of memory lines we propose *Maximize Schedulability* (Lock-MS), a new ILP-based method that considers both intra-task and inter-task interferences. That is, we get the selection of memory lines that provides the lowest overall execution cost (including preloading times) when used in a dynamic cache-locking multitasking system. We show how to model the system with easy to understand path-explicit constraints and then how to transform them into a compact model, which can be solved much faster.

This paper is organized as follows. In Section 2, we review the background and related work. Section 3 presents our path-explicit method for selecting the lines to be locked into the instruction cache. The model compaction is described in Section 4. Section 5 shows experiments comparing several selection procedures, memory architectures and analysis times. Finally, Section 6 presents our conclusions.

2. Related work

Multitask preemptive real-time systems must be schedulable to guarantee their expected operation. That is, all tasks must complete their execution before their deadline. Considering a *fixed priority* scheduler, feasibility of periodic tasks can be tested in a number of ways [3]. Response Time analysis is one of these mathematical approaches, and fits very well as a schedulability test. This approach is based on the following equation for independent tasks:

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \quad (1)$$

where R_i is the response time, C_i is the WCET and T_i is the period of each task i, respectively. It is assumed that tasks are ordered by priority (the lower the i the higher the priority). This equation provides the response time of each task after a few iterations and it has been used in previous studies [4–6]. A task t_i meets its real-time constraints if $R_i \leqslant D_i$, being D_i the deadline of the task.

The WCET of each task (C_i) is not easy to obtain in systems with cache memory. In the literature we find different methods to approach the WCET problem in the presence of caches [7]. These methods can be divided into those that analyze the normal behavior of the cache, and those which restrict its behavior to simplify the analysis.

The first kind of methods try to model each task and system as accurately as possible considering the dynamic behavior of the cache [8–15]. We compare our approach to one of these methods. A conventional cache analysis is very hard, since the worst path depends on the cache outcome, and the cache outcome affects the cost of each path. Due to this complexity, interferences among tasks are not usually considered and tasks are analyzed in isolation. This means that complementary methods are needed to adapt the WCET of each isolated task to multitasking systems. This may be done by further analysis to add the number of inter-task interferences and their cost to the cost of each task [5,6].

In turn, cache-locking methods restrict the cache behavior by using the ability to disable the cache replacement. Having specific contents fixed in cache, the timing analysis is easier, so these methods can afford a full system analysis, i.e., several tasks on a real-time scheduler. Cache-locking techniques can also be divided into *static* and *dynamic cache-locking*.

Static locking methods preload the cache content at system start-up and fix this content for the whole system lifetime so that it never gets replaced [2,16]. Martí Campoy et al. use a genetic algorithm to obtain the selection [16], whereas Puaut and Decotigny propose two low-complexity selection algorithms: one to minimize the utilization (Lock-MU) and another to minimize the interferences (Lock-MI) [2]. Lock-MI is no longer considered in similar studies, since Lock-MU always exhibits a better behavior. None of these three algorithms studies the possibility of worst path changes depending on the selected cache lines. Instead, the worst path is determined only once, assuming an empty cache. Afterward, these algorithms make a selection of lines to be locked, such that they optimize this path. These techniques are also called "single-path analyses" [17] and, as authors say, their approach is non-optimal. Studies comparing Lock-MU and the genetic algorithm approach conclude that their performance is very similar [18]. We use Lock-MU as a static locking reference, thus avoiding the possible dependencies on the initialization parameters that genetic algorithms may present. Also, we compare to a single-cycle fetch system (ideal performance bound).

Cache-locking approaches that disable cache replacements but allow the cache contents to be changed at run-time are known as dynamic locking methods [19–24]. Essentially, these methods differ in how they load the contents and lock the cache. The operating system may be used to change cache contents at context switches by means of a subroutine [19,20]. As an alternative, the content replacement can be launched by the operating system when a task reaches a certain program counter value [21]. Also,

[2] Mainstream Instruction Set Architectures supporting whole cache-locking include, for instance, ARM (ARM940), MIPS (Integrated Device Technology IDT79RC64xxx), Motorola 68K (Freescale Coldfire), Power (Freescale MPC74xx, MPC8540, PowerPC 440 core, e300 core). Other processors support partial cache-locking only (e.g. most ARM, IDT79RC46xx).