Towards a Predictable and High Performance Use of Instruction Caches in Hard Real-Time Systems

Alexis Arnaud Isabelle Puaut
IRISA, Campus universitaire de Beaulieu,
35042 Rennes Cédex, France
Email: aarnaud/puaut@irisa.fr

Abstract

Cache memories have been widely used in order to bridge the gap between high speed processors and relatively slow main memories. However they are a source of predictability problems. A lot of progress has been achieved to model caches, in order to determine safe and precise bounds on (i) tasks' WCETs in the presence of caches; (ii) cache-related preemption delays. An alternative approach to cope with cache memories in real-time systems is to lock their contents so as to make memory access times and cache-related preemption delays entirely predictable. In this paper, we focus on instruction caches and we describe the state of the art in the so-called static cache locking technique and its related algorithms. Then benefits and problems with this approach are discussed. Eventually refinements and an enhanced technique, dynamic cache locking, are sketched.

1 Caches and real-time systems

Introduction. Cache memories have been designed as a solution for the ever-growing discrepancy of speed between processors and relatively slow memory components. These are typically small content associative memories with small access time, which are inserted between the CPU and the main memory and which act as ante-memories. Indeed no change is required in the memory addressing scheme since caches act transparently. They work in such a way that they exploit the spatial and temporal locality of memory reference streams. Therefore at any time cache memories contain memory blocks that are likely to be accessed in the near future. A key property of cache memories is that they improve the average performance of a computer system.

A real-time system is a computer system for which the good functioning is not only defined by the correctness of results, but also by the the dates at which these results are to be produced. More particularly, a hard real-time system requires the exact knowledge of these dates. In order to satisfy this requirement, such a system must use an adequate scheduling policy such as Fixed Priority Preemptive (FPP), and has to be validated. The validation process consists of the computation of the worst-case execution time (WCET) of each task, and then in performing a schedulability analysis on the whole task set. Extensive studies have been done on both latter subjects.

Now cache memories are a source of unpredictability issues. Two phenomena consolidate this fact :

- Intra-task interferences, which occur when a task overrides its own blocks in the cache, due to conflicts.
- Inter-task interferences, which arise in multitasking systems, due to preemptions. These interferences imply a so-called cache-related preemption delay to reload the cache after a task was preempted.

As a consequence, the designer of a hard real-time system may choose not to use cache memories at all, at the cost of over-sizing the system, or may choose to use scratch-pad memories [1], which are basically on-chip static memories. Nevertheless there is a growing demand in the industry, of hard real-time systems with better performance and cheaper hardware. This fact drives to consider processors designed for general purpose computers. As regards the validation process, an important issue here is to cope with the effects of cache memories which are common in these processors.

Current approaches. There are at the present time two categories of approaches to deal with caches in real-time systems. In the first one, *cache analysis*, caches are used without any restriction. Static analysis techniques (cache-aware WCET analysis [11, 10] and schedulability analysis [8]) predict their worst-case impact on the system schedulability.

The second category of approaches consists in using them in a restricted or customized manner so as to adapt them to the needs of real-time systems and schedulability analysis.
Cache partitioning techniques [7, 9] assign reserved portions of the cache to certain tasks in order to guarantee that their most recently used code or data will remain in the cache while the processor executes other tasks. The dynamic behavior of the cache is kept within partitions. These techniques eliminate inter-task interferences, but need extra support to tackle intra-task interference (e.g. static cache analysis) and reduce the amount of cache memory available for each task.
Within the same category, another way to deal with caches in real-time systems is to use *cache locking techniques*, which load the cache contents with some values and lock it to ensure that the contents will remain unchanged. This ability to lock, entirely or partially, the cache contents is available on several commercial processors (among others : Motorola ColdFire MCF5249, Motorola PowerPC 603e, IDT RC64575, ARM 940). The cache contents can be loaded and locked at system start for the whole system lifetime (*static cache locking*), or changed at run-time, like for instance when a task is preempted by another one (*dynamic cache locking*). The key property of cache locking is that the time required to access the memory is predictable.
Throughout the rest of this paper, a cache is assumed to be an instruction cache unless otherwise specified.
We will call *instruction block* or simply *block* a contiguous sequence of instructions that can be mapped into a cache block.

2 Algorithms for static cache locking

It has been proven in [12] that the problem of the optimal placement of contents in a cache memory, in the sense that it minimizes the number of cache misses, is NP-hard. This result is even stronger : unless P=NP, it belongs to the class of extremely inapproximable problems.
As a consequence, the goal of an algorithm for selection of cache contents must be to minimize a quantity which is not the number of cache misses, and its design must take into account a heuristic. At the present time two solutions have been proposed and are presented below. In the following, we consider a set of periodic tasks which run within the framework of a hard real-time system comprising one level of instruction cache in which the contents may be locked on a per-line basis.

2.1 A genetic algorithm for selection of cache contents

In this first solution [5], the optimization problem consists in minimizing the cache-aware response time [4] and at the same time choosing the cache contents so as to get the best possible performance. It is solved by means of a genetic algorithm. Each individual contains only one chromosome, which describes the state of the blocks of the whole task set. Each gene of a chromosome is a bit, and represents the state (locked or not) of an individual block. The fitness function is a weighted mean of the response times of the tasks, where, for each task, the weight depends on the state of its blocks. The response time of each task is obtained through the Cache-aware Response Time Analysis (CRTA). For each individual, its probability of being selected for crossover depends on its degree of validity, which is a function of the number of locked blocks and the average response time. The population generation process is repeated a prior-defined number of times.

2.2 Pseudo-polynomial algorithms for selection of cache contents

In this solution [13], two greedy algorithms have been designed. Both have a pseudo-polynomial complexity. From the task periods and access statistics of instruction blocks along the worst-case execution path of each task, each algorithm computes a heuristic and selects the cache contents accordingly, so as to minimize a well chosen cache-aware metric, and thus to improve the task set schedulability.
The first algorithm, named Lock-MU (for Minimize Utilization) tries to minimize the cache-aware CPU utilization [2] of the task set. The second one, known as Lock-MI (for Minimize Interferences) tries to minimize, for every task of the system, its cache-aware response time.

2.3 General remarks

Benefits. Static cache locking is an advantageous approach with many respects. As compared with a system which does not use any instruction cache, the worst-case and average-case performances of a system with a locked cache may be improved. Since the cache contents are locked, neither intra-task nor inter-tasks conflicts occur, so that the technique of estimation of the WCET of a task is facilitated. The schedulability analysis is simplified as well, since the cache related preemption delay is constant. Furthermore power consumption is reduced because fetching contents from the cache consumes less power than fetching it from the main memory, and no cache line replacement policy is enabled.

Influence of the size of the task set. If the ratio between the size of the task set and the size of the cache memory is very high, only a very small fraction of the task set will benefit from the cache, which will lead to an important deterioration of the performance of the system. In this regard, as the size of the task set grows, the performance (with respect to either the worst-case or the average-case) with a locked cache will be asymptotically the same as if there were no cache at all. The static cache locking approach thus lacks some scalability.

Average-case execution time (ACET) issues. The algorithms presented above select some instruction blocks so as to minimize cache-aware metrics such as response time or CPU utilization rate when, for each task, its worst-case execution path is taken. Now depending on the algorithmic structure of a task, there may exist one or many feasible execution paths.
Consider a simple program P in which only two paths may be taken, namely the worst-case execution path W and the alternate execution path A. For each path π, $c(\pi)$ represents its execution time and $p(\pi)$ the probability of taking π. Furthermore assume that the cache is locked with contents chosen along the path W, and note $\lambda(\pi)$ the gain obtained on the path π from locking the cache (W and A may share common memory locations). The function λ reaches a maximum for the path W. Then the WCET of P is $C = c(W) - \lambda(W)$, whereas its ACET is $\overline{C} = (c(W) - \lambda(W))p(W) + (c(A) - \lambda(A))p(A)$. From this, we see that the improvement of the average-case performance not only depends on the contents of the cache, but also on the probability that the worst-case path W be taken, and on the *distance* between W and A. Here the distance is defined as the number of memory locations by which W and A differ.

Locality. Another important task property is locality. In most of the programs developed in the industry, memory reference streams tend to exhibit regions with spatial and temporal locality patterns. As previously mentioned, cache memories are designed so as to benefit from this phenomenon. In both approaches presented above for selecting instruction blocks to be locked in the cache, locality is not considered at all.

3 Current work

Stability. In the previous section, we showed the influence of the probability of taking the worst-case execution path on the improvement of the average-case performance of a task. This probability can be thought as a stability factor. More precisely, let T be a task and W its worst-case execution path. W may provide several sub-paths with various stability factors. The exact identification of these sub-paths by means of static analysis appears to be intractable. With regards to the static cache locking, a proposed approach is to consider the set M of different memory locations referenced on W. T is run with various sets of inputs. Then, with help of the corresponding execution traces, each time a memory location $M[i]$ is referenced both on W and on an execution trace, a counter attached to $M[i]$ is incremented. Eventually frequencies are computed on M for all $0 \leq i \leq N$. They define approximate stability factors.
This way, when an algorithm must pick up blocks among those of the whole task set, it would be possible to give a slight advantage to blocks which have a high stability factor. Both the worst and the average cases would benefit from such a policy. As regards the average case, the reasons have already been given above. In the worst case, as there would be a higher rate of stable blocks locked into the cache, there would be globally more cache hits, and consequently a better usage of the cache. Moreover the benefits from the approach presented in the section 2 would be kept intact.

Dynamic cache locking. Because of the issues regarding the size of the task set, the static cache locking technique shall not work well with real life programs which are increasingly complex. Even if a large enough cache memory could be provided so as to keep using static cache locking, a fair amount of scalability could be a requirement for some embedded systems.
An evolution of the static technique is the dynamic cache locking technique. As in the static case, the decisions concerning the contents of the cache are taken before the system startup. But here, the contents are changed at some chosen points, according to the advancement of the tasks. There are two possibilities: at a given date, either the cache is assigned to a single task, or it is assigned to a bunch of tasks, as in the static cache locking technique. There is an ongoing work based on the first possibility and on an evolution of the notion of approximate stability factor given in the former paragraph. As in the static case, the goal is to minimize a metric such as the response time of each task.

Spatial and temporal locality. Some work has already been performed on the analysis of locality [6] or the definition of a locality metric [3, 14]. A good definition of a locality metric would provide a way to identify which instruction blocks are best candidates for assignment into an instruction cache, and to design better heuristics for the algorithms which select the contents of the cache.

Architectural refinements. While the model considered in the present paper assumes only one level of cache mem-

ory, most of the systems nowadays provide two or three levels. Since access time grows with the cache level, instruction blocks should be assigned in such a manner that the better their qualities (e.g. stability or locality), the lower the cache level they will occupy.
In the cache locking technique, a cache memory could be replaced with a scratch-pad memory, which furthermore would consume even less power than a locked cache, mainly because of issues related with associativity. But we argue that cache memories are not only more widespread, but also that cache memories allow for a better flexibility. A scratch-pad memory does not act as an ante-memory. Generally, the respective ranges of addresses of a scratch-pad memory and the main memory are separated and share the same address space. Thus transparency is lost and a relocation mechanism is necessary. If tasks run directly in physical memory, some modifications have to be done on the tasks.

Heterogeneous systems. Some systems incorporate tasks with various degrees of real-time requirements. A way to deal with such systems is to allow, for example, static cache locking for critical interrupt handlers, dynamic cache locking for other hard-real time jobs, and normal cache operations for soft real-time jobs.

References

[1] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory : A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign, CODES, Estes Park (Colorado)*, May 2002.

[2] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[3] Mark Brehob and Richard Enbody. An analytical model of locality and caching. Technical Report MSU-CSE-99-31, Departement of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, August 1999.

[4] J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 204–212. IEEE Computer Society, June 1996.

[5] Marti Campoy, A. Perles Ivars, and J. V. Busquets Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, London, United Kingdom, December 2001.

[6] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE transactions on computers*, 45(11):1319–1326, 1996.

[7] D. B. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS89)*, pages 229–237, Santa Monica, California, USA, December 1989.

[8] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[9] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, pages 213–227, Washington - Brussels - Tokyo, June 1997. IEEE.

[10] Thomas Lundqvist and Per Stenstrm. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, November 1999.

[11] F. Mueller. Timing analysis for instruction caches. *Real-time systems*, 18(2):217–247, May 2000.

[12] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, Portland, Oregon, 2002. ACM Press.

[13] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *23rd IEEE International Real-Time Systems Symposium*, Austin, TX, USA, December 2002. IEEE.

[14] Dee A.B. Weikle, Sally A. McKee, Kevin Skadron, and Wm.A. Wulf. Caches as filters : A framework for the analysis of caching systems. In *Proceedings of the 3rd Grace Hopper Celebration of Women in Computing*, September 2000.