Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems

Isabelle Puaut
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
e-mail: puaut@irisa.fr

Abstract

Cache memories have been extensively used to bridge the gap between high speed processors and relatively slow main memories. However, they are source of predictability problems and need special attention to be used in hard real-time systems. A lot of progress has been achieved in the last 10 years 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 caches in real-time systems is to statically lock their contents so as to make memory access times and cache-related preemption times entirely predictable. This paper attempts to evaluate qualitatively and quantitatively the pros and cons of both classes of methods.

1 Caches and real-time systems

Extensive studies have been performed on schedulability analysis to guarantee timing constraints in hard real-time systems. Schedulability analysis methods assume that task worst-case execution times (WCETs) are known. While many schedulability analysis methods consider that the cost of task preemption is zero to simplify the analysis, some methods account for task preemption costs (e.g. manipulation of task queues, cache-related preemption delays).

Caches are small and fast buffer memories used to speed up the memory accesses. They contain memory blocks that are likely to be accessed by the CPU in the near future. Although the caches are a very effective means of speeding up the memory accesses in the average case, they are a source of predictability problems, due to intra-task and inter-task interferences. *Intra-task* interferences occur when a task overrides its own blocks in the cache due to conflicts, while *inter-task* interferences arise in multitasking systems due to preemptions. The inter-task interferences imply a so-called *cache-related preemption delay* to reload the cache after a task is preempted.

Caches raise predictability issues in hard real-time systems because they are designed to speed up the system *average case* performance rather than the system *worst case* performance which is of prime importance in hard real-time systems. As a consequence, the designers of hard real-time systems may choose not to use cache memories at all, or may choose to use on-chip static RAM - scratchpad memories – instead of caches [2]. The simple approach consisting in assuming that every access to memory results in a cache miss, causes the tasks WCETs to be largely overestimated, which may cause the schedulability analysis to fail while the system may actually be feasible. The main issue is then to estimate tasks WCETs and cache-related preemption delays in a safe but not overly pessimistic manner.

Two classes of approaches, described hereafter, can be used to deal with caches in real-time systems.

Cache analysis methods. A first class of approaches to deal with caches in hard real-time systems is to use them without any restriction, and resort to *static analysis* techniques to predict their worst-case impact on the system schedulability.

At the intra-task level, static WCET analysis techniques have been extended to predict the impact of cacheing on the WCETs of the tasks. They achieve a classification of the memory accesses regarding the instruction or data caches (e.g. hit when it can be proved that the access always results in a cache hit, miss otherwise). Techniques to predict the worst-case task behavior regarding the instruction cache can use data-flow analysis on each task control flow graph [12], abstract interpretation [1], integer linear programming techniques [10], or symbolic execution [11].

At the inter-task level, work has been undertaken to obtain safe and precise estimates of the cache-related preemption delay [9]. In this work, at every possible preemption point, the blocks that will be used by each task after that point are determined by static analysis, thus avoiding considering that the whole memory accessed by the task has to be reloaded in the cache after a preemption.

Cache partitioning and cache locking. A second class of approaches to deal with caches in real-time systems is to use 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 [8, 5, 14] assign reserved portions of the cache (partitions) 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 the 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.

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 in order to ensure that the contents will remain unchanged [6]. This ability to lock cache contents is available on several commercial processors. The cache contents can be loaded and locked at system start for the whole system lifetime (*static cache locking*), or changed during the system execution, like for instance when a task is preempted by another one (*dynamic cache locking*). The key property of static cache locking is that the time required to access the memory is *predictable*.

Schedulability analysis for systems with caches. Some schedulability analysis methods (Rate Monotonic Analysis – RMA, Response Time Analysis – RTA) have been extended to cope with cache-related preemption delays in [3] and [4] respectively. They add the parameter γ_i, the cache-related preemption delay, to the formulas in charge of verifying the system feasibility (e.g. $\sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i} \leq n(2^{\frac{1}{n}}-1)$ in RMA for n tasks of period P_i and WCET C_i).

2 Cache analysis vs static cache locking

In the following, we give some elements that allow to choose between using statically locked caches or using the dynamic features of the caches, which imposes to use cache analysis techniques to bound accurately tasks WCETs and cache-related preemption delays. A static cache locking strategy with a frozen cache contents for *all tasks* is considered hereafter.

2.1 Qualitative comparison

Static cache locking is attractive from several point of views. First of all, it improves the system performance compared to a system that does not use caches, with respect to both average and worst-case system performance.

In addition, with static cache locking, the time required to perform a memory access is *predictable* (it is either a hit or a miss depending on whether the value is locked in the cache or not). While WCET analysis is still required, it alleviates the need for using complex cache analysis techniques for computing WCETs and cache-related preemption delays, and results in more simple WCET analysis tools. In particular, it eliminates the issue of integrating cache analysis techniques with the analysis techniques for the other architectural features (pipelines, branch prediction, etc).

Static cache locking can also be used when no cache analysis method can apply, due for instance to non-deterministic or poorly documented cache replacement strategies (e.g. pseudo-random replacement policies).

Another important benefit of static cache locking is that the technique addresses both intra-task and inter-task interferences, which is unique among the cache management techniques presented above. Concerning inter-task interferences, since in static cache locking schemes the cache blocks are statically partitioned among tasks, the cache-related preemption delay is null, or is constant and equal to the time required to reload the processor prefetch buffer if the processor is equipped with such a architectural feature. This low cache-related preemption delay is particularly important for large caches (see section 2.2).

However, statically locking the contents of instruction caches reduces the amount of cache memory available for each task. In addition, it raises the issue of selecting the cache contents. As we are interested in hard real-time systems, the main objective of the cache selection algorithm is to improve the worst-case system behavior according to some of the metrics used by schedulability analysis methods, such as CPU utilization or interferences between tasks. The main issue is then to avoid performing an exhaustive search of all possible cache contents, which would require an untractable computation cost. For instance, if every cache block can contain 4 program lines, checking the feasibility of the system with all possible cache contents would require 4^B feasibility tests, with B the number of cache blocks. This complexity led [6] to select a genetic algorithm for the selection of the cache contents and [13] to base the selection of cache contents on actual traces of the system execution.

Another potential benefit of static cache locking, although not proved yet by any study, is that it can easily apply to data caches, unified caches of multi-level caches.

2.2 Quantitative comparison

Since the primary focus in hard real-time systems is to prove that all deadlines are met, the key performance metric to be considered when comparing cache management schemes is the *worst-case* performance of the system. In this section, we compare the worst-case performance of a small task set made of periodic tasks (table 1 shows the task

Task name	Description	Code size (Bytes)	WCET-miss	Period
qurt	Computation of roots of quadradic equations	1824	21474	59697
minver	Matrix inversion	4320	36701	70098
jfdctint	JPEG integer implementation of the forward DCT	3440	29324	127559
fft1	FFT (Fast fourier transform) Cooly-Turkey algorithm	3620	115152	601093

Table 1. Task set characteristics

Asso \ Size		**512B**	**1KB**	**2KB**	**4KB**	**8KB**	**16KB**
1	Locking	+0.779	+0.665	+0.576	+0.517	+0.517	+0.517
	Analysis	+0.547	+0.413	+0.382	+0.388	+0.388	+0.388
2	Locking	+0.775	+0.658	+0.518	+0.459	+0.420	+0.420
	Analysis	+0.638	+0.439	+0.382	+0.388	+0.388	+0.388
4	Locking	+0.779	+0.623	+0.485	+0.418	+0.375	+0.368
	Analysis	+0.788	+0.609	+0.414	+0.389	+0.388	+0.388
8	Locking	+0.775	+0.622	+0.491	+0.415	+0.368	+0.368
	Analysis	-1.039	+0.771	+0.578	+0.421	+0.389	+0.388
16	Locking	+0.777	+0.602	+0.485	+0.414	+0.368	+0.368
	Analysis	-1.011	-1.035	+0.744	+0.585	+0.421	+0.389
32	Locking	+0.777	+0.602	+0.484	+0.412	+0.368	+0.368
	Analysis	-1.076	-1.007	-1.004	+0.750	+0.585	+0.421

Table 2. Compared worst-case performance of static cache locking and cache analysis

set characteristics[1]) using a state of the art *cache analysis technique* with its worst-case performance obtained using *static cache locking*.

The static cache locking algorithm implemented [13] selects the contents of the statically locked cache according to the knowledge of the tasks memory accesses, obtained using simulation. It locks the mostly used program lines of the tasks in the cache, in order to minimize the worst-case CPU utilization ($\sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i}$, with C_i, P_i and γ_i denoting respectively the WCET, period and cache-related preemption delay of task i)

We compare the worst-case performance of this task set with the one obtained through the use of a state of the art cache analysis technique based on F. Mueller's work on static cache simulation (see [12, 7] for details). The Heptane tree-based WCET analysis tool [7] has been used to compute WCETs. No attempt is made here to bound the cache-related preemption delay γ_i precisely (it is assumed that all program lines of a given task have to be reloaded after a preemption, with a maximum of N reloads where N is the number of cache lines).

The worst-case system performance of the task set is given in Table 2. Each cell indicates whether the task set is feasible or not according to CRTA [4] (Response Time Analysis enhanced with the knowledge of cache-related preemption delays γ_i , that are null for static cache locking and considered maximum for cache analysis). A '+' sign means that the task set if feasible, whereas a '-' sign means that it is not. The CPU utilization of the task set is also given in each cell. These two pieces of information are given for different cache sizes (Bytes), degrees of associativity, and this with and without static cache locking.

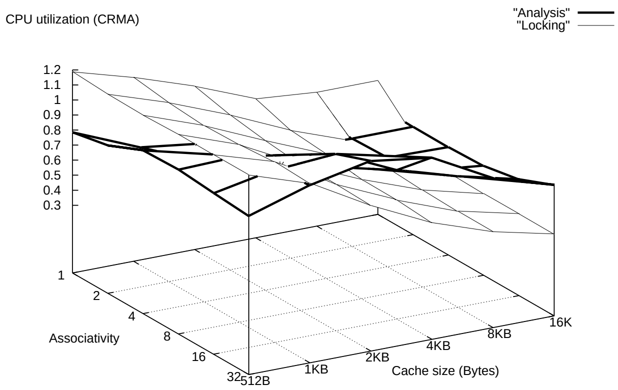

Figure 1. Worst-case CPU utilization

Figure 1 depicts the CPU utilization obtained on the task set from the contents of table 2. It compares the CPU utilization obtained when using cache locking and static cache

[1] In the table, delayes are expressed in number of processor cycles for a MIPS processor with a simplified timing model. WCET-miss denotes the WCET of the task assuming that all instructions cause a cache miss.

analysis. It can be noted that for a given degree of associativity, the performance of both static cache locking and static cache analysis increases with the cache size, because of the decrease of the number of conflicts for cache blocks. However, the performance increase of static cache locking is higher than the one of static cache analysis when the cache size increases. This is because the cache-related preemption delay increases linearly with the cache size for the static cache analysis method, whereas it stays constant for the static cache locking method.

For a given cache size, the performance of static cache locking scales better than the one of static cache analysis with an increasing degree of associativity W. Indeed, static cache locking takes benefit of the increasing degree of associativity to eliminate both intra-task and inter-task interference, which explains that the CPU utilization increases with W. In contrast, the static cache analysis method we have used does not scale well with W.

3 Open issues

The key benefits of static cache locking is to make the time required to perform memory accesses predictable, and to be a unified technique to take into account both intra-task and inter-task conflicts for cache blocks. This class of techniques alleviates the need for using complex static analysis techniques for computing WCETs and cache-related preemption delays. In addition, it can be applied in situations where static cache analysis cannot be used at all (*e.g.* when the instruction cache has a non deterministic or non documented cache replacement policy). While algorithms already exist for selecting the contents of statically locked caches [6, 13], we think that further work is required:

- to study their performance on larger real (non synthetic) benchmarks, in particular in task sets whose size is much larger than the cache size. For large programs, a possible direction is to explore more dynamic cache locking strategies (for instance, to select different contents of the locked cache changed at statically-defined points in order to cope with the tasks dynamic behavior while staying predictable)
- to study the impact of statically locked caches on the system average case performance
- to study the applicability of static cache locking techniques to data/unified/multi-level caches
- to address implementation issues on actual embedded processors
- to compare the use of statically locked caches with the use of on-chip static RAMs (benefits wrt predictability, issues to be addressed)

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS'96, Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*, pages 51–66. Springer, September 1996.

[2] O. Avissar, R. Barua, and D. Stewart. Heterogeneous memory management for embedded systems. In *Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, Atlanta, GA, USA, Nov. 2001.

[3] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 1994.

[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 Press, June 1996.

[5] J. V. Busquets-Mataix and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Proc. of the 9th Euromicro Workshop of Real-Time Systems*, pages 56–63, Toledo, Spain, June 1997.

[6] M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. Static use of locking caches in multitask premptive real-time systems. In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, London, UK, Dec. 2001.

[7] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[8] 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, Dec. 1989.

[9] C. G. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6), June 1998.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, IEEE Computer Society Press, Dec. 1996.

[11] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, Nov. 1999.

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

[13] I. Puaut. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. Submitted to publication - available on demand, May 2002.

[14] J. E. Sasinowski and J. K. Strosnider. A dynamic programming algorithm for cache/memory partitioning for real-time systems. *IEEE Transactions on Computers*, 42(8):997–1001, Aug. 1993.