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

Isabelle Puaut
INSA/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 describes work in progress aiming at evaluating 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.
- *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 caching 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

* Proc. of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems, Vienna, Austria, June 2002

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 [9], 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 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 cache locking is that the time required to access the memory is *predictable*.

Schedulability analysis for systems with caches. Some schedulability analysis methods have been extended to cope with cache-related preemption delays. They add the parameter γ_i, upper bound on the cache-related preemption delay, to the formulas in charge of verifying the system feasibility.

In [3], Rate Monotonic Analysis (RMA) is extended to cope with cache-related preemption delays. The utilization of a set of periodic tasks that takes the cache-related preemption delays into account is introduced (see equation 1 below).

$$U = \sum_{i=1}^{n} \frac{C_i + \gamma_i}{P_i} \quad (1)$$

In the equation, n is the number of tasks. C_i and P_i are the WCET and period of task number i. For static priority systems with priorities assigned along the rate monotonic policy, a sufficient condition given in equation 2 below can then be used to verify the system schedulability [3].

$$U \leq n(2^{\frac{1}{n}} - 1) \quad (2)$$

Response Time Analysis (RTA) has been extended by Busquets-Mataix et al [4] to take cache-related preemption delays into account, leading to the exact schedulability condition named CRTA. The principle of CRTA, for a task T_i, is to consider the interferences produced by the execution of the higher priority tasks on an increasing time window w_i^n. The response time R_i of task T_i is the fixed point of the sequence given in equation 3 below, with γ_i the cache-related preemption delay and $hp(i)$ the set of tasks that have a higher priority than T_i.

$$\begin{aligned} w_i^0 &= C_i \\ w_i^{n+1} &= C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil (C_j + \gamma_j) \rightarrow R_i \end{aligned} \quad (3)$$

These series converge when $\sum_{j \in hp(i) \cup \{i\}} \frac{C_j}{P_j} \leq 1$. The response time R_i of task T_i can then be compared against its deadline to determine the schedulability of T_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 together with static 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 the 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 an architectural feature. This low cache-related preemption delay is particularly important for large caches (see section 2.2).

Finally, implementing cache locking turns out to be a light task once the contents of the locked cache are selected. No modification of the compilation process is required to implement static cache locking. In particular, the addresses of values (instructions/data structures) need not be modified, contrary to schemes that use static on-chip RAM to speed up memory accesses. To be implemented, static cache locking only requires to execute a small routine at the system start-up to load the contents of the cache with the selected values and lock the cache so that its contents remain unchanged during the whole system execution.

However, statically locking the contents of caches reduces the amount of cache memory available for each task. In addition, it raises the issue of selecting the cache contents. Since 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, for a direct-mapped cache, if up to 4 program lines can me mapped onto a given cache block, 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, or to unified caches or to 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 using a state of the art *cache analysis technique* with its worst-case performance obtained using *static cache locking*. In this performance evaluation, we focus on *instruction caches* only.

2.2.1 Experimental setup

Target architecture and simulator. The target architecture considered in the experiments is the simplified MIPS processor used by the Nachos operating system. Nachos[1] is a simple operating system for an emulated MIPS CPU, designed for teaching purposes. Nachos has been extended for the purpose of this study with an instruction cache with blocks of 16 bytes, and with a prefetch buffer of 16 bytes (4 MIPS instructions) to speed up sequential access to instructions. The instruction cache can be parametrized so that the cache size ranges from 512 bytes to 16Kbytes, and so that the associativity degree W ranges from 1 (direct-mapped cache) to 32 (set associative cache). The timing model considered for the processor is very simple: an instruction is assumed to execute in $t_{hit} = 1$ clock cycle in the case of a cache hit, and $t_{miss} = 10$ clock cycles otherwise.

Static cache locking. The algorithm designed to select the contents of the locked cache (see [13] for more details) applies to task sets made of *periodic* tasks. It aims at optimizing the task set schedulability, by minimizing the CPU utilization (as defined in equation 1, $U = \sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i}$).

As the worst-case execution path required to compute the WCET C_i is not known unless the contents of the locked cache are known, the minimization of the utilization U is achieved thanks to the knowledge of an *actual* execution path for the tasks, which is not necessarily the *worst-case* path. This execution path is obtained through a simulation of the execution of the tasks using Nachos.

On the considered architecture, when static cache locking is used, the cache-related preemption delay γ_i is constant and equal to the delay required to refill the processor prefetch buffer ($\gamma_i = t_{miss}$).

[1] http://www.cs.washington.edu/homes/tom/nachos/

	Formulas for WCET computation: $W(S)$
$S = S_1; ...; S_n$	$WCET(S) = WCET(S_i) + ... + WCET(S_n)$
$S =$ if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
$S =$ loop$(tst)S_1$	$WCET(S) = maxiter * (WCET(Test) + WCET(S_1)) + WCET(Test)$ where $maxiter$ is the loop maximum number of iterations.

Table 1. Simplified equations for tree-based WCET computation

Task name	**Description**	**Code size** (Bytes)	**WCET-miss**	**Period**
qurt	Computation of roots of quadratic 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 2. Task set characteristics

WCET analysis and static cache analysis. Worst-case execution times (WCET) are obtained using the Heptane tree-based WCET analysis tool [7]. Heptane computes WCETs through a bottom-up traversal of the syntax tree of the analyzed programs. Table 1 gives a flavor of the formulas used in Heptane to compute WCETs (these formulas are overly simplified since they assume constant and context-independent execution times for the leaves of the syntax tree).

Heptane includes hardware modeling capabilities so as to estimate safely but precisely the WCETs on architectures with instruction caches, pipelines and simple branch predictors. Here, Heptane's pipeline and branch prediction modeling modules have been switched off since our focus is on instruction caches only.

The technique used in Heptane to estimate the instruction cache behavior is based on F. Mueller's so-called *static cache simulation* [12]. The cache analysis technique computes abstract cache states (representation of all possible cache contents considering all possible execution paths in the program), computed using data-flow analysis on the program control flow graph. Abstract cache states are then used to classify the instructions according to their worst-case behavior regarding the instruction cache (e.g. *hit* when it is certain that there are no conflict for a cache block, *miss* when no better classification can be found, and intermediate categories when there are conflicts inside loops). In the case of set-associative caches, when a given instruction appears in different blocks of the same set of an abstract cache state, different categories can be obtained for the different instances of the instructions. In this situation, in order for the analysis to be safe, the instruction is classified with the most pessimistic category.

Heptane has been modified so as to offer the possibility to replace the cache analysis module by a module that takes into account the presence of locked caches. This new module classifies instructions into two categories: *miss* and *hit*. An instruction is classified as a *hit* if it is locked in the instruction cache, and is classified as a *miss* otherwise.

When Heptane is configured to be used with its cache analysis capabilities, no attempt is made 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).

Task set. The worst-case performance evaluation has been achieved on a small real (non synthetic) task set, whose characteristics are given in table 2.

The table gives for every task: its name, a short description, the size of its code in bytes, its WCET assuming that all instruction fetches cause a cache miss, and its period. The delays in the table are expressed in number of processor cycles. The periods of tasks have been selected so that the CPU utilization ($\sum_{i=1}^{n} \frac{C_i+\gamma_i}{P_i}$) equals 1.3 (i.e. the task set is not feasible if no instruction cache is used).

2.2.2 Worst-case performance analysis

The worst-case system performance of the considered task set is given in Table 3. Each cell indicates whether the task set is feasible or not according to CRTA (equation 3, detailed in [4]). A '+' sign means that the task set if feasible, whereas a '-' sign means that it is not. The CPU utilization

Asso \ Size		512B	1KB	2KB	4KB	8KB	16KB
1	Locking	-1.188	-1.073	+0.936	+0.774	+0.740	+0.740
	Analysis	+0.785	+0.605	+0.550	+0.565	+0.567	+0.567
2	Locking	-1.174	-1.039	+0.881	+0.699	+0.551	+0.516
	Analysis	+0.833	+0.701	+0.588	+0.575	+0.567	+0.567
4	Locking	-1.169	-1.022	+0.844	+0.678	+0.506	+0.395
	Analysis	+0.935	+0.806	+0.687	+0.620	+0.577	+0.567
8	Locking	-1.177	-1.030	+0.823	+0.644	+0.495	+0.393
	Analysis	+0.943	+0.954	+0.813	+0.718	+0.623	+0.577
16	Locking	-1.177	-1.052	+0.825	+0.641	+0.489	+0.393
	Analysis	+0.921	-0.997	-1.025	+0.840	+0.721	+0.623
32	Locking	-1.176	-1.044	+0.817	+0.625	+0.487	+0.393
	Analysis	+0.904	-1.031	-1.113	-1.057	+0.842	+0.721

Table 3. Compared worst-case performance of static cache locking and static cache analysis

of the task set (as defined in equation 1) is also given in each cell. These two pieces of information are given for different cache sizes (Bytes), degrees of associativity, and this with static cache analysis (label *Analysis* in the table) and with static cache locking (label *Locking* in the table).

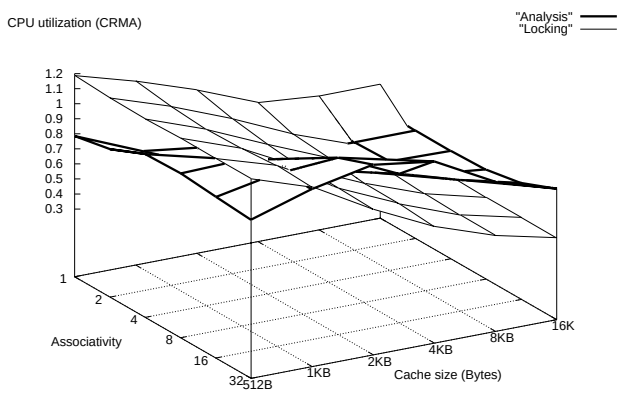

Figure 1. Worst-case CPU utilization

Figure 1 depicts the CPU utilization obtained on the task set from the contents of table 3. It compares the CPU utilization obtained when using cache locking and static cache analysis.

Performance for small cache sizes. It can be noted from the contents of figure 1 and table 3 that for small cache sizes, static cache analysis performs better than static cache locking, both from the standpoint of the task set feasibility and from the standpoint of CPU utilization. This comes from the fact that for small caches, there is a high degree of intra-task and inter-task conflicts for every cache block. By construction, the static cache analysis used in the WCET analyzer is not sensitive to inter-task interferences because tasks are considered separately. Thus, the WCET of tasks with cache analysis is lower than with cache locking. Moreover, the comparatively lower WCETs of cache analysis come with a higher cache-related preemption delay. However, the impact that this higher cache-related preemption delay has on the system schedulability and CPU utilization is negligible for small caches.

Impact of the cache size. 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. For instance, for a 4-way associative cache, the task set exhibits better (i.e. lower) CPU utilization when using static cache locking than when using cache analysis for caches larger than 8KB, whereas static cache locking performs better for caches smaller than 8KB. 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.

Impact of the degree of associativity. 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. This comes from the *pessimistic* way the instructions are classified (see § 2.2.1).

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)

Acknowledgements

Thanks to David Decotigny (IRISA) and Jörn Schneider (Saarland University) for pointing out errors in earlier drafts of this paper.

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 and D. Decotigny. 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.