On Using Locking Caches in Embedded Real-Time Systems*

A. Martí Campoy[1], E. Tamura[2], S. Sáez[1],
F. Rodríguez[1], and J. V. Busquets-Mataix[1]

[1] Departamento de Informática de Sistemas y Computadores,
Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022 Valencia, Spain
{amarti, ssaez, prodrig, vbusque}@disca.upv.es
[2] Grupo de Automática y Robótica,
Pontificia Universidad Javeriana – Cali, Colombia
eutamo@doctor.upv.es

Abstract. Cache memories are crucial to obtain high performance on contemporary processors. However, they have been traditionally avoided in embedded real-time systems due to their lack of determinism. Unfortunately, most of the techniques to attain predictability on caches are complex to apply, precluding their use on real applications. This work reviews several techniques developed by the authors to use cache memories in "real" embedded real-time systems, with the ease of use in mind. Those techniques are based on a locking cache, which offers a very predictable behaviour. Both static and dynamic use are proposed as well as the algorithms and methods required to make the schedulability analysis using two different scheduling policies. Also proposed is a genetic algorithm that finds, within acceptable computational cost, the sub-optimal set of instructions that must be preloaded in cache. Finally, a set of statistical analyses compares the locking cache versus a conventional one.

Keywords: cache memories, embedded real-time systems, genetic algorithms, predictability, schedulability analysis, performance evaluation, execution time, response time

1 Introduction

Embedded systems are composed of a combination of hardware and software components which perform specific functions in host systems, which range from domestic appliances to space explorers. The vast majority of processing elements manufactured worldwide are used in such systems. In some cases, embedded systems need to satisfy stringent timing requirements. Hence, they also may be Real-Time systems, in which the correctness of the system depends not only

* Work partially supported by Ministerio de Educación y Ciencia, Dirección General de Investigación under project DPI2003-08320-C02-01.

on the logical result of computations, but also on the time at which the results are produced.

Embedded Real-Time Systems is a very exciting and expanding field, whose applications are found in command and control systems, process control, automated manufacturing. In every case, they typically control the environment in which they operate and they need to guarantee the response times. To cope with the increasing complexity, many embedded real-time systems use modern microprocessors, which provide a higher throughput. Contemporary microprocessors include cache memories in their memory hierarchy to increase system performance. General-purpose systems benefit directly from this architectural improvement, but for embedded real-time systems, their inclusion raises the complexity when analysing task set schedulability. In fact, using cache memories presents two problems. The first problem lies in estimating the Worst Case Execution Time, *WCET*, due to intra-task or intrinsic interference. Intra-task interference occurs when a task removes its own instructions from the cache due to conflict and capacity misses. When the task tries to execute those removed instructions, cache misses increase the execution time of the task. This way, the delay caused by the cache memory interference must be included in the WCET calculation. The second problem is to estimate the task response time due to inter-task or extrinsic interference. Inter-task interference occurs in preemptive multitask systems when a task displaces the working set of any other task from the cache. When the preempted task resumes execution, a burst of cache misses increases its execution time. This effect, called cache-refill penalty or cache-related preemption delay must be considered in the schedulability analysis, since it situates task execution time over the precalculated WCET. Modelling cache behaviour is very complex, like described in several proposals [3], [8], [7], [6], [2]. Thus, several alternatives to conventional caches have been proposed. One of these alternatives is the use of locking caches.

2 Locking Cache Basics

Several processors include a cache memory with the ability to lock its contents, thus precluding its replacement when the processor fetches new instructions. The use of locking caches in embedded real-time systems offers several advantages:

– Intrinsic interference is eliminated, and extrinsic interference is bounded and can be estimated in advance. This makes cache behaviour very predictable, allowing a simple analysis, even when other architecture improvements are used, since memory access delays are constant. This is an improvement over other alternatives like SMART [5] and others that do not fully remove interferences and still demand complex analyses.
– Necessary hardware is nowadays present in several commercial processors, and only minor hardware modifications are mandatory in order to get the best performance.
– In several cases, the use of locking caches presents about the same or better performance than that obtained when using a conventional cache.

– The use of locking cache is transparent to programmers, since he/she does need to neither include any special instructions nor use additional tools to write the applications.

The only disadvantage when using locking caches is that system performance depends on selecting the instructions loaded and locked in cache. This selection must be carefully accomplished and presents some degree of complexity.

3 Static Use of Locking Caches

The main goal of using statically locking caches is its full predictability and thus, the simplicity when estimating execution and response times [10]. In this case, the locking cache is loaded with a well-known set of instructions before the execution begins, and the cache remains unmodified during system operation. Although locking caches are present in several processors, some minor modifications in these architectures are needed in order to get full predictability and the best possible performance:

– Cache can be totally locked or unlocked. When cache is locked, there are no new tag allocations.
– Cache can be loaded using a cache-fill instruction, selecting the memory block to load it.
– There exists a one-cache-line size buffer to temporarily store those instructions not selected to be loaded into cache, thus improving its sequential access. The penalty incurred for executing instructions to be loaded in this buffer is the same as those executing from cache.

The WCET of tasks may be easily estimated using the timing analysis presented in [13]. The effect introduced by the cache inclusion is then reduced to know which instructions are loaded and locked in cache and which not. Thus, since there are no replacements in cache memory and the set of instructions to be locked is selected by the system designer, the WCET is easily estimated.

Regarding response time of tasks, it can be estimated using Cached Response Time Analysis, *CRTA*, [2], an extension to Response Time Analysis, *RTA*, [1] for fixed priority, *FP*, scheduled systems. CRTA is based in an iterative equation (1) where the cache effect is incorporated in parameter γ_j. This parameter represents the time required to refill the cache after each preemption. When a locking cache is used statically, only the temporal buffer changes during preemptions, so in the worst case the value of γ_j is the time needed to reload the temporal buffer, one cache miss.

$$w_i^{n+1} = C_i + B_i + \sum_{\forall\, j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j)\,. \qquad (1)$$

When the priority of tasks is dynamically assigned, as it happens with an Earliest Deadline First, *EDF*, scheduler, the schedulability analysis is accomplished using the Initial Critical Instant, *ICI*, analysis proposed in [12]. This schedulability test does not consider any cache penalty due to preemptions, so the effect of cache memories must be included in this analysis. Two questions arise when cache effect wants to be considered: the time needed to reload the cache; and the number of preemptions a task suffers. The first question has an easy answer. When a locking cache is statically used, only the temporal buffer must be reloaded. The second question is more difficult to answer in systems with dynamic priorities. It is quite easier however to determine the number of preemptions a task originates when an EDF scheduler is used: these preemptions can only occur on task arrivals. Therefore, a task generates a preemption when it arrives or does not generate any preemption at all.

Since the ICI test is not based upon the individual times of tasks but rather upon the global system utilisation, the cache penalty (reload of temporal buffer) can be accounted for to the preempting task, instead of incorporating this delay in the preempted task. Thus, equations (2) and (3) show the resulting ICI test equations, where the only modification is to add the time needed to reload the temporal buffer (T_{miss}) to the WCET of each task. By making use of functions $G(t)$ and $H(t)$ it is possible to derive the value of the initial critical instant, R, by resolving the recurrence formula given in equation (4) until $R_{i+1} = R_i$.

$$G(t) = \sum_{i=1}^{n} (C_i + T_{miss}) \times \left\lceil \frac{t}{P_i} \right\rceil . \tag{2}$$

$$H(t) = \sum_{i=1}^{n} (C_i + T_{miss}) \times \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor . \tag{3}$$

$$R_{i+1} = G(R_i), R_0 = 0 . \tag{4}$$

4 Dynamic Use of Locking Caches

Dynamic use of locking cache is proposed with a single objective: getting better performance than that obtained through the static use, and at the same time, keeping a high degree of predictability [11]. The operation of dynamic use is quite similar to the one proposed in the static use: loading and locking in a cache memory a previously selected set of instructions. However, in dynamic use, the cache contents change in well known instants of time: every time a task begins or resumes execution, the cache memory is flushed and reloaded with a set of instructions belonging to the new scheduled task. Once the instructions are loaded, the cache is locked until a new task is dispatched for execution. This way, each task may use all the available cache space in order to improve its execution time, in clear contrast with static use, where all tasks must share the cache. In order to operate as desired, hardware and software requirements must be met. First, the processor must offer instructions to unlock and flush the cache. In addition, the operating system must store the list of instructions (addresses) to load in cache for each task; finally, the scheduler must include a small loop to load the cache every time a new task is scheduled. In this scenario, WCET is estimated in the same way that when cache is statically used, since intra-task interference does not exist. However, the estimation of response time of tasks must consider the effect of reload cache after preemptions, and computing this effect is not easy because tasks may suffer two kinds of interference: direct interference or indirect interference. Direct interference means that a task increases its response time because it is forced to reload its own instructions that were previously removed during preemption. Indirect interference means that a task increases its response time because executing any other higher priority tasks increases its response time, due to its own extrinsic interference.

The value of direct-extrinsic interference is the time a task needs to load and lock its instructions in the cache. The value of indirect-extrinsic interference is the time other higher priority task needs to load and lock its instructions in the cache. Since response time analysis must consider the worst-case scenario in order to provide an upper bound of tasks' response time, the maximum possible increment of time must be taken into account for each preemption. Equations (5) and (6) show the cache refill penalty and CRTA equation respectively. $time\_to\_load_z$ is the time a task needs to load its instructions, and γ_j^i is the cache refill penalty for a task τ_i preempted by a task τ_j.

$$\gamma_j^i = \max_{j<z\leq i} (time\_to\_load_z) \,. \tag{5}$$

$$w_i^{n+1} = C_i + B_i + \sum_{\forall\, j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times (C_j + \gamma_j^i) \,. \tag{6}$$

When a dynamic scheduler as EDF is used, the sequence in which the tasks are activated is unknown. This means that a task may be preempted by any other task in the system, but it also means that the preempting task may be preempted at the same time by any other task. That is, the number of tasks that may produce indirect interference has no limit. This way, the value of cache-refill penalty due to indirect interference may be the time to reload the cache of any task in the system but the preempted. Thus, the cache refill penalty for any task will be the maximum from the *time_to_load* of all system's tasks, every time, and every preemption. Since the time needed to reload the cache may significantly vary between tasks, considering this scenario will produce a high overestimation when computing the response time of tasks and the system utilisation. Therefore, dynamic use of locking cache is not suitable for dynamic schedulers, due to the impossibility of getting accurate analysis results.

5 Selecting Contents for the Locking Cache

The increase of performance due to the use of cache memories is very significant; hence, embedded real-time systems must take advantage of it. The architecture of a locking cache guarantees determinism, but not performance. In order to achieve both goals, i.e., a fully predictable cache and a performance similar to that provided by a conventional cache, the instructions to be locked must be carefully selected. It is not easy however to find an algorithm that select blocks to load and lock in cache in a straight way. In preemptive, multitasking systems, the execution time of tasks depend on the execution time of higher priority tasks. In addition, indirect interference in dynamic use causes that the response time of tasks depends on the time needed to reload the cache contents. This way, cache contents must be selected considering not the isolated tasks, but all of the tasks interacting in the system. Exhaustive search, including branch and bound, presents an intractable computational cost, since the number of possible solutions is huge. In addition, since the problem is not monotonic, algorithms like hill climbing are not useful. Genetic algorithms, proposed in [4], performing a randomly directed search, can be used in this problem, finding a sub-optimal solution within an acceptable computational time. Two versions of a genetic algorithm [9], one for static and the other for dynamic use, have been developed. The algorithm evaluates a set of possible solutions using a fitness function to sort them. New solutions are created by combining the best individuals of the previous generation, and the process is repeated a fixed number of times. Since the block is the minimum unit of information that can be transferred from main memory to cache, the algorithm provides the set of blocks to be locked, rather than its individual instructions. It also brings an estimation of the WCET of each task executing in a locked cache with the chosen set of blocks, and the response time of all tasks considering the estimated WCET using the locking cache as given by equations (1) and (6). The main disadvantage of the genetic algorithm is its temporal cost, whose execution takes between four and six hours and it may take up to twelve hours when solving some problems. But, on the other hand, it offers an interesting advantage, because several fitness functions may be used to sort the solutions, thus guiding the algorithm to improve the performance in the way that the system designer is most interested on: minimising system utilisation, maximising task slacks, or tuning the response time of tasks.

6 Experimental Results

Predictability and performance of locking cache have been evaluated using a large set of experiments. Around 30 systems have been used. Each experiment is composed of a set of tasks, ranging from three to eight tasks. Tasks used in experiments are artificially created to stress the proposed cache scheme. A simple tool is used to create tasks. The tool requires the main parameters of every task, such as the number of loops and nesting level, its size, loops size, the number of if-then-else structures and their respective sizes. Task period is hand-defined to make the system schedulable, and the task deadline is equal to its period. The workload of any task may be a single loop, if-then-else structures, nested loops, streamlined code, or any mix of these. The code size for a task may be large (up to 32 Kbytes) or short (lower than 1 Kbyte). More than two hundred experiments had been accomplished. Each experiment is simulated using direct-mapped, two-set associative, four-set associative and fully associative caches, with cache sizes ranging from 1 Kbyte to 64 Kbytes. For all cases, line size is 16 bytes (four instructions) and in most of the cases, the task set footprint is bigger than the cache size; furthermore, in some cases, just one task may require more space than the cache can provide. Fetching any instruction from main memory takes 10 cycles, while fetching any instruction from cache (or temporal buffer) takes just 1 cycle. For each experiment, the response time of each task is estimated using the genetic algorithm, then simulated in a locking cache using the blocks selected by the genetic algorithm, and finally it is simulated in a conventional cache to evaluate performance and predictability.

First results concern predictability and accuracy of analysis methods. For a Fixed Priority Scheduler, FPS, response time of tasks (RT_e) as estimated by the genetic algorithm is compared with the response time of tasks obtained by simulating its execution with a locking cache (RT_{sl}). For the EDF scheduler, the system utilisation (U_e) as estimated by the genetic algorithm is compared with the system utilisation obtained by simulating its execution with a locking cache (U_{sl}).

Figure 1 show the cumulative frequency polygons of error between the estimated and simulated results. For an FPS, the error (or overestimation) is defined as $e_{fps} = (RT_e/RT_{sl}) - 1$; in the case of EDF the following formula is used: $e_{edf} = (U_e/U_{sl}) - 1$.

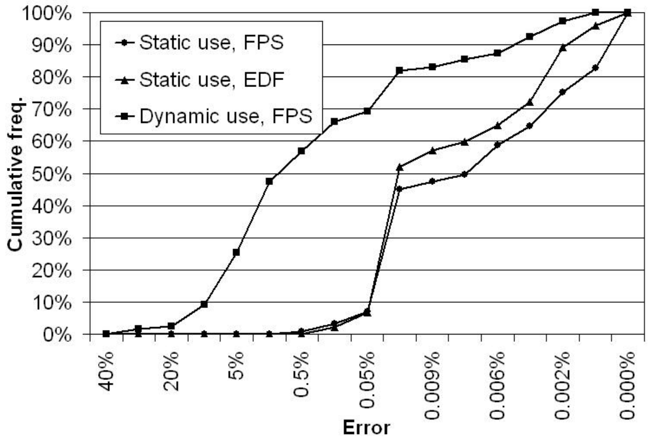

Fig. 1. Cumulative frequency polygon of error between estimated and simulated response time

From Figure 1 it can be seen that for static use and FP scheduler, the error is less than 1% in the whole set of experiments. Furthermore, in more than 90% of the cases, the error is lower than 0.05% and it is below than 0.01% in around 50% of the cases. The results obtained when using a statically locked cache with an EDF scheduler look very similar to those achieved with the FP scheduler. However, it is possible to observe a slightly greater error; it is 0.01% in around 40% of the cases. This is due to the excessively conservative assumption that states that every task causes a preemption when it is activated. Yet, since cache refill penalty is extremely low, error increases in an almost negligible way. Figure 1 also shows the dynamic use of locking cache with an FP scheduler and shows an overestimation, which is higher than that in the two previous cases since taking into account the worst case is inherent to indirect preemptions. In 10% of the cases, error fluctuates between 10% and 30% but it falls down to 1% for more than 50% of the cases. Certainly, there exists a high variability in the error obtained.

Figure 2 illustrate results concerning performance and show the cumulative frequency polygons of gain/loss of performance, P, when comparing utilisation using a conventional (simulated) cache, U_c, and the utilisation using a locking cache as estimated by the genetic algorithm, U_e. Here, $P = U_c/U_e$. A result less than one indicates that the utilisation using the locking cache is higher than the utilisation using the conventional cache, thus losing performance. On the other hand, results higher than one mean that the use of locking caches offers, not just determinism, but also a performance gain.

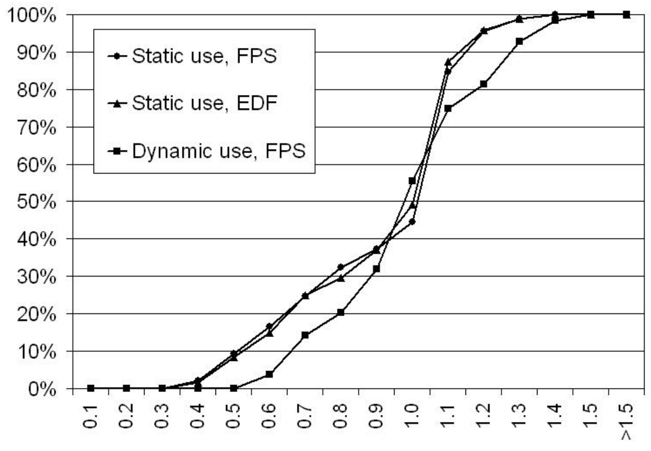

Fig. 2. Cumulative frequency polygon of gain/loss of performance of locking cache in front of conventional cache

In Figure 2, it is also possible to note that for static use and FP scheduler in more than 60% of the cases, there are no significant losses in performance (those in which the ratio is above 0.9). The same conclusions can be drawn from Figure 2 for static use and EDF scheduler. As can be seen in Figure 2, in the case of the dynamic use and FP scheduler, about 70% of the experiments do not demonstrate significant losses in performance; besides that, in 20% of the cases, there is a significant gain in performance (above 1.2) when dynamic use is employed.

7 Conclusions and Future Work

The use of locking caches in embedded real-time systems has proved to be very useful, since it exhibits a highly predictable behaviour, thus facilitating the schedulability analysis and, at the same time, offering a performance analogous to that provided by a conventional cache, which on the other hand, is hard to incorporate into the real-time system analysis.

Moreover, dynamic use of locking cache beats any previous proposal using cache memory in an embedded real-time system:

- In contrast to alternative proposals to conventional caches, the locking cache completely removes intrinsic interference while extrinsic interference is tightly bounded. Other approaches have some level of unpredictability, thus requiring more complex models and analyses to estimate both the execution and the response times.
- Even though using locking cache poses performance losses in some cases when compared to using conventional caches, none of the existing proposals is able to offer a tightly precise estimation, thus resulting also in a performance loss in practical terms.

Albeit it might seem that there are no further possibilities in using locking caches in embedded real-time systems, there still exist some paths to follow. In all of them, the main goal is to increase the performance of the locking cache. This can be done as follows:

- By reducing the time required reloading cache contents in dynamic use of locking cache. This has a twofold effect. First, minimising the time required to reload cache obviously minimises the execution times. In addition, the overestimation of the response times is minimised, which in practical terms is equivalent to a performance increase, since the designer may fine-tune the system in a better way. This reduction can be accomplished by means of a memory hierarchy like those proposed in [14] in which the cache memory can be locked on a per-line basis and include flags to reflect the line lock status for the blocks pertaining to the current executing task. The memory hierarchy also needs an extra, dedicated SRAM to store the locking state information for the whole task set plus some simple, easy to add hardware for proper operation.
- In addition, since it has been found that the performance of the outcome of the genetic algorithm can be very dependant on the fitness function used, the genetic algorithm may provide different fitness functions to satisfy the system designer needs by allowing him/her to optimise the utilisation, the slack, or by trying to find a trade-off solution in between.
- Finally, the genetic algorithm is being parallelised to be executed in a Linux cluster with a message-passing environment by using the homogeneous "island" approach, in which several loosely-related sub-populations are processed by different processing elements to speed up the calculations.

References

1. Audsley, A.N., Burns, A., Richardson, M., Tindell, K.: Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal, **8** (1993) 284-292

2. Busquets-Mataix, J.V., Wellings, A.J., Serrano-Martin, J.J., Ors-Carot, R., Gil, P.: Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In: Proc. of the Eighth Euromicro Workshop on Real-Time Systems. IEEE Computer Society Press, Los Alamitos (1996) 271-276
3. Healy, C.A., Arnold, R.D., Mueller, F., Harmon, M.G., Walley, D.B.: Bounding pipeline and instruction cache performance. IEEE Trans. Comput. **48** (1999) 53-70
4. Holland, J. H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge (1992)
5. Kirk, D.B.: SMART (Strategic Memory Allocation for Real-Time) cache design. In: Proc. of the 10th IEEE Real-Time Systems Symposium. IEEE Computer Society Press, Los Alamitos (1989) 229-237
6. Lee, C.-G., Hahn, J., Seo, Y.-M., Min, S.L., Ha, R., Hong, S., Park, C.Y., Lee, M. C., Kim, S.: Enhanced analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In: Proc. of the 18th IEEE Real-Time Systems Symposium (RTSS '97). IEEE Computer Society Press, Los Alamitos (1997) 187-198
7. Li, Y.-T.S., Malik, S., Wolfe, A.: Cache modeling for real-time software: beyond direct mapped instruction caches. In: Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS '96). IEEE Computer Society Press, Los Alamitos (1996) 254-263
8. Lim, S.-S., Bae, Y.H., Jang, G.T., Rhee, B.-D., Min, S.L., Park, C.Y., Shin, H., Park, K., Moon, S.-M., Kim, C.S.: An accurate worst case timing analysis for RISC processors. IEEE Trans. Softw. Eng. **21** (1995) 593-604
9. Martí Campoy, A., Pérez Jiménez, A., Perles Ivars, A., Busquets Mataix, J.V.: Using genetic algorithms in content selection for locking-caches. In: Proc. of the IASTED International Symposia Applied Informatics. Acta Press, Innsbruck (2001) 271-276
10. Martí Campoy, A., Perles Ivars, A., Busquets Mataix, J.V. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. Proceedings of the IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the 22nd IEEE Real-Time Systems Symposium), London, UK, December 2001.
11. Martí Campoy, A., Perles Ivars, A., Busquets Mataix, J.V. Dynamic Use of Locking Caches in Multitask, Preemptive Real-Time Systems. Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control, Elsevier Science, Barcelona, Spain. July 2002.
12. Ripoll, I., Crespo, A., Mok, A.: Improvement in feasibility testing for real-time tasks. Journal of Real-Time Systems. **11** (1996) 19-40
13. Shaw, A.C.: Reasoning about time in higher-level language software. IEEE Trans. Softw. Eng. **15** (1989) 875-889
14. Tamura, E., Rodríguez, F., Busquets-Mataix, J.V., Martí Campoy, A.: High Performance Memory Architectures with Dynamic Locking Cache for Real-Time Systems. In: Proc. of the Work-In-Progress session of the 16th Euromicro Conference on Real-Time Systems. Available as Technical Report from the University of Nebraska-Lincoln, Department of Computer Science and Engineering (TRUNL-CSE-2004-0010), (2004) 1-4