Proceedings of the

15th International Conference on Real-Time and Network Systems

RTNS'07

LORIA, Nancy, France

29-30 March 2007

http://rtns07.irisa.fr

Towards Predictable, High-Performance Memory Hierarchies in Fixed-Priority Preemptive Multitasking Real-Time Systems

E. Tamura
Grupo de Automática y Robótica
Pontificia Universidad Javeriana – Cali
Calle 18 118–250, Cali, Colombia
eutamo@doctor.upv.es

J. V. Busquets-Mataix and A. Martí Campoy
Departamento de Informática de
Sistemas y Computadores
Universidad Politécnica de Valencia
Camino de Vera s/n., 46022 Valencia, España
{vbusque, amarti}@disca.upv.es

Abstract

Cache memories are crucial to obtain high performance on contemporary computing systems. However, sometimes they have been avoided in real-time systems due to their lack of determinism. Unfortunately, most of the published techniques to attain predictability when using cache memories are complex to apply, precluding their use on real applications. This paper proposes a memory hierarchy such that, when combined with a careful pre-existing selection of the instruction cache contents, it brings an easy way to obtain predictable yet high-performance results. The purpose is to make possible the use of instruction caches in realistic real-time systems, with the ease of use in mind. The hierarchy is founded on a conventional instruction cache based scheme plus a simple memory assist, whose operation offers a very predictable behaviour and good performance thanks to the addition of a dedicated locking state memory.

1 Introduction

Contemporary computing systems include cache memories in their memory hierarchy to increase average system performance. In fact, cache memories are crucial to obtain high performance when using modern microprocessors. While trying to minimise the average execution times, the contents of the cache memories vary according to the execution path. General-purpose systems benefit directly from this architectural improvement; however, minimising average execution times is not so important in real-time systems, where the worst-case response time is what matters the most. Thus, due to their lack of determinism, sometimes cache memories have been avoided in fixed-priority preemptive multitasking real-time systems: when they are incorporated in such a system, in order to determine the memory hierarchy access times as well as the delays involved in cache contents replacement it is necessary to know what its contents are.

Using cache memories in fixed-priority preemptive multitasking real-time systems presents two problems. The first problem is to calculate the *Worst-Case Execution Time* (*WCET*), due to intra-task or intrinsic interference. *Intrinsic interference* occurs when a task removes its own instructions from the instruction cache (*I-cache*) due to conflict and capacity misses. When the removed instructions are referenced again, cache misses increase the execution time of the task. This way, the delay caused by the I-cache interference must be included in the WCET calculation. The second problem is to calculate the *Worst-Case Response Time* (*WCRT*) due to inter-task or extrinsic interference. *Extrinsic interference* occurs in preemptive multitask systems when a task displaces instructions of any other lower priority tasks from the I-cache. When the preempted task resumes execution, a burst of cache misses increases its execution time. Hence, this effect, called cache-refill penalty or *Cache-Related Preemption Delay* (*CRPD*) must be considered in the schedulability analysis.

This work proposes

- a memory hierarchy that provides high performance coalesced with high predictability. The solution is to be centred on instruction fetching since it represents the highest number of memory accesses [15];
- the required schedulability analysis for such hierarchy; and
- some evaluation results and its analysis.

Results show that

- the proposed memory hierarchy is predictable and simple to analyse;

- its performance exceeds that of the dynamic use of locking cache as given in [10]; and
- in many cases, its performance is about the same than that obtained when using a conventional instruction cache.

The remainder of the paper is organised as follows. Section 2 introduces the problem and summarises some of the solutions found in the literature. Section 3 describes the proposed memory hierarchy, its requirements, a functional description of its operation and the schedulability analysis. Section 4 assesses the proposed memory hierarchy by comparing it with the dynamic use of locking cache as given in [10]. First predictability and prediction accuracy are examined by comparing estimated and simulated worst-case response times. Performance is evaluated by measuring the worst-case processor utilisation. Some concluding remarks are given in Section 5.

2 Rationale

In order to guarantee that every task in the task set meets its deadline, real-time system designers may opt for three different approaches:

- Use the memory hierarchy in a conventional manner.
- Use the memory hierarchy in a real-time systems suitable manner.
- Use a real-time systems aware memory hierarchy.

Each approach will be briefly summarised according to three different perspectives: architectural viewpoint, implementation viewpoint and, run-time support viewpoint.

2.1 The memory hierarchy is used in a conventional manner.

When using cache memories in a conventional way, the memory hierarchy is the same used in any conventional system with cache memories; therefore, regarding implementation and run-time support, there is no need to implement any additional hardware or software modules. Instead, the real-time system designer does his/her best to determine whether each memory reference causes a cache hit or a cache miss. This is done by using static analysis techniques. Some of the techniques used for WCET calculation are data-flow analysis [13, 22], abstract interpretation [1], integer linear programming techniques [6], or symbolic execution [7]; to tackle the WCRT estimation data-flow analysis is also used. Unfortunately, the complexity of static analysis techniques may preclude their use in practical applications.

2.2 The memory hierarchy is used in a real-time systems suitable manner.

An alternative to fully exploit the inherent performance advantage of cache memories while achieving predictability is to work with unconventional memory hierarchies. In this case, instead of conventional cache memories, the real-time designers favour the use of either locking caches [10, 18, 25, 2, 17] or scratchpad memories [26, 27]. On the one hand, locking caches are caches with the ability to lock cache lines to prevent its replacement; blocks are loaded into the locking cache and then they are locked. They are accessed through the same address space as the main memory. On the other hand, scratchpad memories are an alternative to I- or D-caches (data caches). They are small and extremely fast SRAM memories (since they are usually located on-chip); they are mapped into the processor's address space and are addressed via an independent address space that must be managed explicitly by software.

Regarding implementation, in both cases, during the design phase it is necessary to choose for every task in the task set which instruction blocks will be either loaded and then locked into the locking cache or copied into the scratchpad memory. The number of selected blocks per task must not exceed the capacity of either the locking cache or the scratchpad memory (selecting which information is copied into a scratchpad is very close to deciding which information has to be locked into a locking cache). Once the blocks are chosen, it is possible to know how much time it would take to fetch every instruction in the whole task set; therefore, the access time to the corresponding memory hierarchy is thus predictable. At compile time, the assignment of memory blocks to either the locking cache or the scratchpad has to be handled by hand or automatically using a compiler and/or a linker. However, since scratchpad memories are mapped in the processor's memory space, explicit modifications in the code of tasks may be required to make control flow and address corrections.

To improve the execution performance of more than one task (as is desirable in a fixed-priority preemptive multitasking real-time system), the contents of either the scratchpad or the locking cache memory should be changed at run-time (dynamic use). Thus, in both cases, the subset of blocks selected for every task should be loaded during system execution by a software routine, which is executed each time the real-time system designer judges convenient. Transfers to and from scratchpad memories are under software control while for locking caches this is transparent. While a task is not preempted, it is necessary to ensure that the contents of either the scratchpad or the locking cache will remain unchanged. This way, extrinsic interference is eliminated while intrinsic interference can be bounded. In [10] using locking instruction caches is proposed to cope with both extrinsic and intrinsic interferences; in [25], the use of locking D-caches is proposed to enhance predictability by inserting locking/unlocking instructions: the cache is locked whenever it is not possible to statically determine whether the memory references a datum inside the cache or not. In several cases, the dynamic use of locking I-caches effects the same or better performance than using a conventional I-cache [10]. In [27] by using scratchpads performance gains comparable to that of caches are also obtained. However, since the amount of scratchpad memory available is often small compared to the total amount of cache memory available, intuitively, it is reasonable to think that for task sets with big tasks the scratchpad memory approach may obtain lower performance than the cache memory approach.

No matter

1. which mechanism is used to trigger the execution of a small software routine to either load blocks into the locking cache (at the scheduler level, as proposed originally in [10] or via debug registers by raising exceptions when the program counter reaches specified values [2]) or copy blocks to the scratchpad memory; and,
2. the location of the software routine (e.g., in main memory or even in a scratchpad memory),

the execution of the aforementioned software routine demands valuable processor cycles. Since this execution time must be added to the task's WCRT, the overhead introduced when using either locking caches or scratchpad memories in a fixed priority multitasking real-time system may have severe consequences on performance.

2.3 The memory hierarchy is real-time systems aware.

A third option is to design more predictable memory hierarchies. A memory hierarchy for fixed-priority preemptive multitasking real-time systems must implement mechanisms which in some way address the effects of

- intrinsic interference: it must prevent that the contents of the cache are overwritten by the same task;
- preemption: by allowing the preempting task to overwrite the contents of the cache; and
- extrinsic interference: it must allow that the contents of the cache are restored when the preempted task resumes execution.

To deal with extrinsic interference, some of the approaches use cache partitioning techniques, which allocate portions of the cache to tasks via hardware (I-cache [5], D-cache [14]), software (by locating code and data so they will not map and compete for the same areas in the cache) [28, 12] or a combination of hardware and software [19, 3]. Notice that the technique proposed in [5] introduces unpredictability for blocks that go to the shared pool.

To improve predictability, [4] proposes to extend the cache hardware and to introduce new instructions to control cache replacement (kill or keep cache blocks).

In [24], a custom-made cache controller assigns partitions of set associative caches to tasks so that extrinsic interference is eliminated; cache partitions are assigned to tasks according to their priorities by using a prioritised cache: each partition is assigned dynamically at run time; higher priority tasks can use partitions that were previously allocated to lower priority tasks. A partition allocated to a higher priority task cannot be used for a lower priority task unless the former notifies the cache controller to release the partitions it owns (which is done when the task is completely over). Therefore, it might be possible that the highest priority tasks consumes the whole cache memory and jeopardises the lowest priority tasks response times.

The work presented in this paper is a refinement of previous work [23] and proposes the use of an I-cache and additional hardware information to influence the I-cache replacement decision. This "cache replacement policy" provides a mechanism to increase predictability (time-determinism) without degrading performance, making it suitable for use in fixed-priority preemptive multitasking real-time systems. In this approach, the subset of selected blocks for each task and the instants in which I-cache flushing takes place are fixed: Every time a task begins or resumes its execution, the I-cache is flushed and then it is gradually reloaded with selected blocks as the instructions belonging to the task to be dispatched are being fetched. The selected blocks are inhibited from being replaced until a new context switch takes place. This way, the access time to the memory hierarchy is predictable and on the other hand, each task may use all the available I-cache space in order to improve its execution time.

In contrast to other approaches, the proposed memory hierarchy does not need any software to load the selected blocks into the I-cache at run time and hence it does not introduce penalties in the task's WCRT.

3 Memory hierarchy architecture

Efficient operation of the memory hierarchy requires an efficacious, automatic, on-demand storage control method that frees the software from explicit management of memory addressing space. Furthermore, the resulting architecture should not introduce any additional delays and be as open as possible by using generic components.

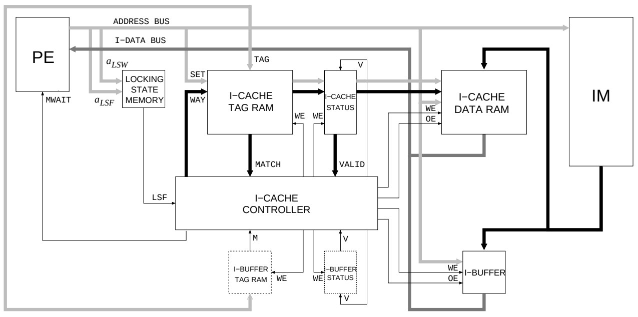

Figure 1. Proposed memory hierarchy

3.1 Description

Figure 1 sketches an architecture that pursues these goals. As can be seen, the figure does not embody any locking I-cache; it resembles a system for a conventional I-cache. There are however three noteworthy differences:

- There is an extra, dedicated, very fast SRAM memory, the *Locking State Memory* (*LSM*), located to the right of the *Processing Element* (*PE*). Its role is to store the status of every instruction block (the *Locking State*, *LS*) in the *Instruction Memory* (*IM*), thus providing a mechanism to discriminate which blocks must be loaded into the I-cache and hence a way to allow for automatic, on-demand loading of the selected instruction blocks. In other words, instead of locking selected blocks into an instruction locking cache, the same effect can be attained by avoiding loading into the I-cache unselected blocks.

- There is also an *Instruction Buffer* (*I-buffer*), with size equal to one cache line, located below the I-cache controller. Having an I-buffer is not essential, rather it is more of a performance assist: its purpose is to take advantage of the sequential locality for those blocks that should not be loaded into the I-cache. Since the I-buffer catches and holds previously used instructions for reuse, it might also contribute with temporal locality by providing look behind support (via the boxes drawn with dashed lines in the bottom part of the figure).

- There is also a subtle difference in the control bits of the I-cache with respect to a locking cache: since locking state information is stored into the LSM, locking status bits are not required.

3.2 Performance requirements

The main goal of the memory hierarchy is to provide deterministic yet high-performance response times.

In order to achieve determinism, each time a task τ_i is dispatched for execution, its corresponding subset of previously selected blocks, SB_i, is loaded into the I-cache as the PE fetches them. Once loaded, the selected blocks must remain in the I-cache and must not be overwritten as long as task τ_i is either not preempted by other, higher priority tasks or it finishes. This policy, which is applied to every task in the task set, eliminates intrinsic interference since the task is not allowed to remove any block previously loaded into I-cache, thus contributing to temporal determinism. Furthermore, extrinsic interference is bounded and can be estimated in advance.

Both temporal locality, the tendency to access instructions that have been used recently, and spatial locality, the tendency to involve a number of instructions that are clustered, are essential to performance. Hence, by keeping the SB_i blocks loaded in the I-cache, temporal locality is mainly captured by the I-cache yet spatial locality is also supported. Besides that, the I-buffer captures spatial locality for those blocks not in SB_i, albeit as it was said before, it might also provide some temporal locality.

With respect to timing issues, the goal is to cause minimum overhead during I-cache (re)load: since the LSM is not in the critical path, IM latency remains the same. LSM access time however must be in the order of a cache hit time to operate in parallel with the I-cache and its controller. This way, the I-cache inner workings are not affected and hence, its timings remain about the same.

3.3 Storage requirements

Storage requirements for the LS are also very important: space consumption should be low. Regarding cost, the most useful measure is to determine how much memory needs to be added to the system.

The minimum amount of memory required to keep track of each selected block is one bit. Hence, there will be as many bits as the number of blocks in the IM. Each of those bits will store a flag, the *Locking State Flag* (*LSF*), which is used to signal whether the corresponding block should be loaded into I-cache or not. For LS packing purposes, however, it is better to group the information into wider, off-the-shelf, fast SRAM memories. Henceforth assume an 8-bit wide LSM; then, the information for eight blocks (a *parcel*) will be stored in one *Locking State Word* (*LSW*) as shown in Figure 2.

Let L be the I-cache line size in bytes and let b_I be the number of bytes per instruction; then each memory block has L/b_I instructions. Given an IM of depth $d_{IM} = mL$, where m is the number of instruction blocks, the required LSM has a depth, d_{LSM}, equal to $m/8$. Then, the number of instructions, I, that corresponds to each LSW is given by $I = 8L/b_I$.

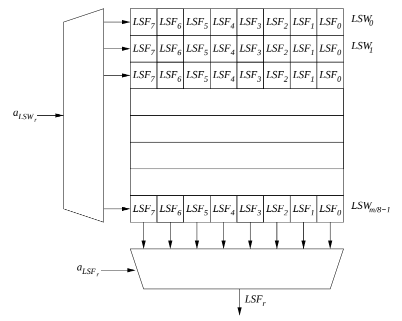

Figure 2. Locking state memory

Let w be the IM width in bits, a be the width of the address bus in bits, K be the degree of associativity and N the number of sets, then S_{IM}, the space required for IM; S_{CM}, the space required for I-cache memory; and S_{LSM}, the space required for LSM, are given (in bits) by:

$$S_{IM} = wmL \quad (1)$$

$$S_{CM} = wLKN + \left(a - \log_2 \frac{L}{N}\right) \times KN + KN \quad (2)$$

$$S_{LSM} = m \quad (3)$$

In the expression for S_{CM}, the first term is related to the I-cache data memory, the second one reflects the space needed for its tag memory and the last one accounts for its status bits (just the valid bits are considered). Notice that lock bits are not necessary since they are grouped into the LSM.

Therefore, the space efficiency, η_s, which measures the fraction of memory dedicated to store LS, can be defined as the ratio of the LSM space, S_{LSM}, to the total amount of memory space:

$$\eta_s = \frac{S_{LSM}}{S_{IM} + S_{CM} + S_{LSM}} \quad (4)$$

3.4 Functional operation

During system design, given a task set, TS, an off-line algorithm selects a subset, SB_{TS}, from the task set instruction memory blocks ($SB_{TS} = \bigcup SB_i, \forall\ \tau_i \in TS$).

The LS associated to TS, LS_{TS}, which reflects the status of every instruction block in TS, must then be loaded into the LSM and it will remain fixed during system execution.

When the system starts operating, the PE must reset the I-cache controller and invalidate all the entries in the I-cache as well as in the I-buffer.

Now, every time that an instruction, I_r, at address a_r is referenced by the dispatched task, τ_i, the LSM needs to be accessed to check the LSW at address a_{LSW_r}. This is the address of the LSW that corresponds to m_r, the memory block that embodies I_r. Hereafter, assume a 32-bit wide instruction size and a byte-addressable IM. Then, a_{LSW_r} is obtained by stripping off the $\log_2 8b_I$ least significant bits of a_r.

Finally, it is necessary to extract LSF_r, the corresponding LSF within LSW_r to drive the `LSF` signal and thus determine whether it is necessary to load m_r in the I-cache. The LSF is indexed by using the 3 bits next to the $\log_2 b_I$ least significant bits of a_r to drive an 8-way multiplexer.

At the same time, the tag for m_r is compared in the I-cache directory thus updating the `MATCH` signal and its corresponding line status is checked via its `VALID` bit. Simultaneously, the data portion of the I-cache is also accessed. Based upon the `LSF`, `MATCH` and `VALID` signals, the I-cache controller may have three possible outcomes:

- The `LSF` signal is `1`, indicating that m_r must be loaded and locked in the I-cache so the I-buffer is disabled; in other words, $m_r \in SB_i$. If the reference causes a miss (because either there is no tag match or the entry is not valid), m_r is loaded from IM into the I-cache, the corresponding tag is updated and its valid bit is set. Afterwards, the I-cache controller, via the `MWAIT` line, signals the PE that the instruction is available so that it can restart fetching.

- The `LSF` signal is `1`, but the reference results in a hit (because the instruction was previously referenced during the current execution). Then, the PE can fetch the instruction from the I-cache without incurring in any further delays.

- The `LSF` signal is `0`, indicating that m_r should not be loaded in the I-cache; in other words, $m_r \notin SB_i$. In this case, the I-cache is disabled and it is necessary to access the IM in order to load m_r in the I-buffer.

Each time a context switch occurs, the scheduler executes an instruction that causes that the entire I-cache contents are purged (its valid bits are reset) and therefore, every line is invalidated; the I-cache controller should also be reset to avoid that it finishes incomplete operations taking place when the context switch happened. Not purging the I-cache might bring better performance but in any case, it is quite difficult to estimate which blocks will remain in the I-cache after several preemptions; furthermore, it is harder to know if those blocks will be used at all once the preempted task resumes execution. Thus, since one of the primary goals is to keep the schedulability analysis simple, it is better to purge the cache on each context switch. Notice however that this may introduce an overestimation in the schedulability analysis.

Using an LSM in the proposed way imposes a constraint: since in a conventional I-cache there is no hardware impediment to replace its lines, the block selection algorithm must guarantee that for any set in the I-cache there will be no conflict misses. Otherwise, selected blocks, which are already loaded, may be overwritten. This might cause some performance improvements, but at the same time, its predictability will deteriorate and hence, the analyses will turn more complex.

Aside from this restriction, it is important to note that the focal feature of the memory hierarchy is the inclusion of the LSM. With the LSM, the proposed memory hierarchy is able to provide a *Virtual Locking I-cache*. Its key advantage is that it uses a conventional I-cache like a locking I-cache. This approach then, takes advantage of the I-cache intrinsic features while at the same time avoids the overhead required to load instructions into the locking I-cache and the explicit manipulation of its locking mechanism.

3.5 Schedulability analysis

The schedulability analysis is done in two steps: in the first step, the WCET of each individual task is calculated assuming that it is the only task in the system but accounting for the intrinsic interference. Subsequently, the effect of the extrinsic interference is considered in the second phase, the calculation of the WCRT.

Task's WCET is estimated by using a *Cache Aware Control Flow Graph, CACFG*, an extended *Control Flow Graph, CFG* [21]. In a CACFG, each memory block is mapped to a cache block and assigned a block number and each *basic block* (i.e., each sequence of instructions with a single entry/single exit point) inside the memory block is mapped to a different vertex. Thus, CACFG models not only the flow control of the task through vertices (as it happens in CFG) but also takes into account the presence of the I-cache by modelling how the task is affected from the point of view of the cache structure.

The WCET of tasks may then be easily estimated considering the execution time of each vertex: Let a task τ_i, with selected vertices $V_i \in SV_i \subseteq SB_i$. The execution time of a vertex depends on the number of instructions inside it, k_{V_i}, and the cache state when the instructions inside the vertex are executed. Since

- in the worst case, SB_i, the subset of selected blocks, and hence SV_i, the subset of its corresponding vertices, will always be loaded on-the-fly by the proposed memory hierarchy each time τ_i executes; and,

- each block, once loaded, will remain in the I-cache as long as task τ_i is not preempted (or it finishes),

it is possible to affirm that, in this particular case, the cache state for τ_i is essentially the same during each of its activations. Thus, the execution times for τ_i's vertices are constant across each execution.

Its WCET can then be estimated assuming that all of the vertices in SV_i are already loaded in the I-cache and then adjust this WCET by accounting for the time required to load SB_i. Hence, if the subset of selected blocks is already loaded in the I-cache and the execution time of any instruction (not including the fetch time) is given by t_I, the WCET for a vertex is given by:

$$k_{V_i} \times (t_I + t_{hit}), \quad \forall\, V_i \in SV_i \tag{5}$$
$$k_{V_i} \times (t_I + t_{hit}) + t_{miss}, \quad \forall\, V_i \notin SV_i \tag{6}$$

and C_i, the WCET for any task can be estimated by applying the approach given in [21]. Notice however that Equation 6 introduces an overestimation in the schedulability analysis whenever there is a control transfer from one vertex to any other vertex that belongs to the same memory block.

Nevertheless, the previous assumption makes necessary to adjust the execution time of those instructions contained in every selected block, B_i. Then, for each selected block B_i not loaded into I-cache, task τ_i will incur in an overhead given by t_{miss} (a compulsory miss).

When estimating the WCET for every task τ_i, the worst case scenario regarding the blocks in SB_i implies loading all of its blocks. Thus, this preemption penalty can be accounted for by adding the term $k_{SB_i} \times t_{miss}$ to the previously calculated WCET:

$$C_i' = C_i + L_{SB_i} \tag{7}$$
$$L_{SB_i} = k_{SB_i} \times t_{miss} \tag{8}$$

where k_{SB_i} is the number of selected blocks for task τ_i. Notice that when using a scratchpad memory or a locking cache in a dynamic way (i.e., by modifying its contents at run time, it is necessary to add an extra term to L_{SB_i}: $\Delta_{SW_{rSB_i}}$, that takes into account the time required to execute the software routine in charge of replacing the corresponding memory.)

WCRT is then obtained by using Equation 9, where the I-cache refill penalty due to extrinsic interference is incorporated in parameter γ_j^i.

$$w_i^{n+1} = C_i' + \sum_{\forall\, \tau_j \in hp(\tau_i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times \left(C_j' + \gamma_j^i\right) \tag{9}$$

Computing γ_j^i 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, previously removed by its preempting tasks. *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 direct and indirect extrinsic interference.

It is hard to know which kind of extrinsic interference a task will suffer during its execution; then, to consider both possibilities, it is safe to use the maximum I-cache refill penalty:

$$\gamma_j^i = \left[\max(k_{SB_j}) + 1\right] \times t_{miss}, \forall\, j \in hp(i) \quad (10)$$

Using the maximum I-cache refill penalty gives a safe, upper bound while keeping the complexity low. This may be somewhat pessimistic: it may happen that not all of the loaded blocks are going to be used before the next preemption. Nevertheless, getting a more precise value in advance will involve complex analyses, since it depends on the number of blocks effectively loaded and the exact preemption instants.

Equation 9 is a recursive equation that is solved iteratively; the resulting WCRT, R_i, is then compared to τ_i's deadline to decide schedulability.

4 Assessing the proposed memory hierarchy

The proposed architecture, when operating in Virtual Locking I-cache mode, is able to guarantee determinism per se (since it is possible to analyse its impact on the WCRT of every task), but system performance strongly depends on the blocks selected to be loaded in the I-cache. Thus, this selection must be carefully accomplished. In fixed-priority preemptive multitasking systems, tasks response times depend on the execution time of higher priority tasks. In addition, indirect interference causes that the response time of tasks depends on the time needed to reload the I-cache contents. Therefore, I-cache contents must be selected considering not the isolated tasks, but all of the tasks in the task set.

Then, the goal is to optimise some temporal metric by selecting a subset of instruction blocks, SB_{TS} from the set of instruction blocks, B_{TS}. Choosing the cache contents in a way that maximises the probability of finding the instructions in cache is a combinatorial problem. In general, the techniques employed to solve combinatorial problems are characterised by looking for a solution from among many potential solutions. Petrank and Rawitz [16] showed that unless $P = NP$ there is no efficient optimised algorithm for data placement or code rearrangement that minimises the number of cache misses. Furthermore, it is not even possible to get close. Therefore, they conclude that the problem pertains to the class of extremely inapproximable optimisation problems and that, consequently, on one hand, it is necessary to use heuristics to tackle the problem, and on the other hand, it is not possible to estimate the potential benefits of an algorithm to reduce cache misses. So, the virtues of a given algorithm must be evaluated by comparing algorithms.

Hence, rather than trying to find only the best (optimal) solution, a good non-optimal (trade-off) solution is sought. Therefore, to solve the problem at hand, it may be a good idea to apply some form of directed search. For this kind of problem, one of the most appealing techniques is using genetic algorithms since they are generally seen as optimisation methods for non-linear functions.

In fact, in [8] a *Genetic Algorithm*, *GA*, has been proposed to solve an equivalent problem. The results presented there show that the use of a genetic algorithm to solve the problem represents a good choice since it provides for each task in the task set, not just the subset of blocks to be loaded, an estimation of the WCET and, the corresponding WCRT considering the estimated WCET, but also because that selection offers good performance. Moreover, results in [9] show that using the genetic algorithm proposed in [8] brings slightly better results than using the pragmatic algorithms given in [18].

In this work, for evaluation purposes, the following cache characteristics are assumed: A direct-mapped I-cache with varying size, a cache line size of 16 bytes (4 32-bit wide instructions); I-buffer is also 16 bytes wide. Fetching an instruction from I-cache or I-buffer takes 1 cycle while fetching an instruction from IM takes 10 cycles. A fixed-priority preemptive scheduler is used in every case. Task priority is assigned according to a Rate Monotonic Policy. Also, notice that in this work, it is assumed that the deadline, D, is equal to the task period, T.

Evaluation results concerning the proposed memory hierarchy must show whether the proposed memory architecture is predictable and if there is any performance loss when using the proposed memory hierarchy (LSM) in front of using a locking I-cache in a dynamic manner (dLC). Therefore, two kinds of results were evaluated to assess the merits of the proposed memory hierarchy. The first set of results is obtained by using a GA to select blocks and estimate processor utilisation when using those selected blocks with the proposed memory hierarchy (U_{LSMe}). The second set of results is obtained by using the same selected blocks and a modified version of SPIM (the freely available, widely used MIPS simulator) which embodies a cache simulator, to execute one hyperperiod of the task set and thus obtain the simulated processor utilisation (U_{LSMs}).

It is not easy to compare the performance of a real-time system running on different architectures. If the same task set is schedulable in every case, there are many characteristics and metrics useful to compare performance. Furthermore, it is highly desirable to use standard benchmark(s) to evaluate the predictability and performance of the proposed memory hierarchy since it makes possible the comparison with other approaches.

Traditional computing benchmarks are inadequate for characterising real-time systems since they are not de-

Table 1. Main characteristics of task sets and cache sizes

Feature	Minimum	Maximum
Number of tasks	3	8
Task Size	1.6 KB	27.6 KB
Task Set Size	12.5 KB	57.6 KB
Instr. executed per task (approx.)	50,000	8,000,000
Instr. executed per tasks (approx.)	200,000	10,000,000
Cache Size	1 KB	32 KB

signed to exhibit behaviour characteristic of such systems, such as periodic, transient and transient periodic activation/deactivation. On the other hand, there are several proposals for embedded/real-time systems benchmarking. Unfortunately, however, the lack of consensus about using a standard benchmark (to the authors' best knowledge) precludes the use of such proposals given that, in general, they are not easily portable. Moreover, it is necessary to notice that the proposed benchmarks are not targeted to measure cache memory effects in real-time systems since they do not cause preemptions[20].

The 26 tasks sets used in this work come from [10]. The code for each task is synthetic; it does nothing useful but it has a mix of instructions such that it is easy to automatically generate different programs, which is adequate for the purpose: each task may have streamlined code, single loops, up to three nested loops, if-then-else constructs.

Table 1 summarises some characteristics of the task sets and cache sizes employed for evaluation purposes.

4.1 Predictability analysis

To verify how predictable the proposed memory hierarchy is, the GA estimated response time of every task in the task set, R_{LSMe}, was compared with the corresponding response time obtained through the simulation R_{LSMs}.

However, instead of using the individual response times for each task, τ_i, in every task set, Processor Utilisation, a measure that involves the whole TS will be used to illustrate the results in a more compact way:

$$U = \sum_{i=1}^{tasks} \frac{C_i''}{T_i} \qquad (11)$$

where C_i'', the computation time of τ_i includes all cache effects (intrinsic and extrinsic interference); i.e., it includes the time required for τ_i to reload the cache after preemptions:

$$C_i'' = R_i - \sum_{\forall\ \tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j' \qquad (12)$$

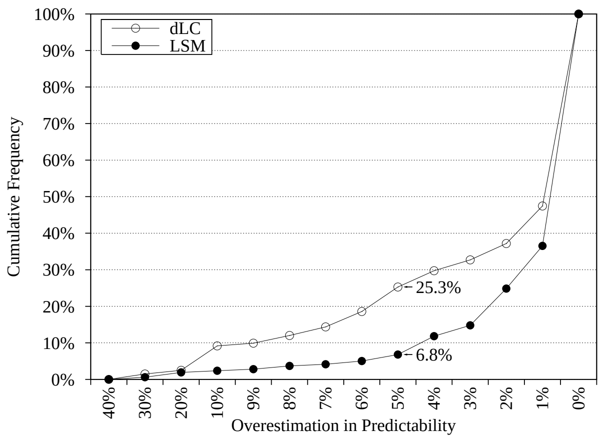

Figure 3. Cumulative frequency curves for the overestimation in predictability

where R_i is the WCRT for τ_i. Since R_i includes not just the CRPD but also the execution time of those tasks with a higher priority than τ_i, it is necessary to deduct the execution time for those tasks.

Then, given the proposed memory hierarchy, the utilisation estimated by the GA (U_{LSMe}), and the utilisation obtained through the simulation (U_{LSMs}), the overestimation in predictability, Ω, is given by $\Omega = U_{LSMe}/U_{LSMs}$. Figure 3 presents the cumulative frequencies for the overestimation when using the proposed memory hierarchy and the dynamic use of locking cache. Cumulative frequencies represent the number of responses in the data set falling into that class or a lower class [11].

The results verify that the proposed memory hierarchy is predictable:

- For every task in the whole set of tasks (676), the estimated response time is always larger than the simulated one ($R_{LSMe} > R_{LSMs}$).

- In the same vein, for every task set, the estimated utilisation is always larger than the one obtained through the simulation ($U_{LSMe} > U_{LSMs}$);

- Furthermore, as can be seen in Figure 3, the proposed memory hierarchy (LSM) provides better predictability than that obtained with the dynamic use of locking cache (dLC). It can be observed that when using the proposed memory hierarchy, the overestimation in utilisation is greater than or equal to 5% in less than 7% of the cases. On the other hand, when employing the locking cache in a dynamic way, the overestimation in utilisation is greater than or equal to 5% in around 25% of the cases.

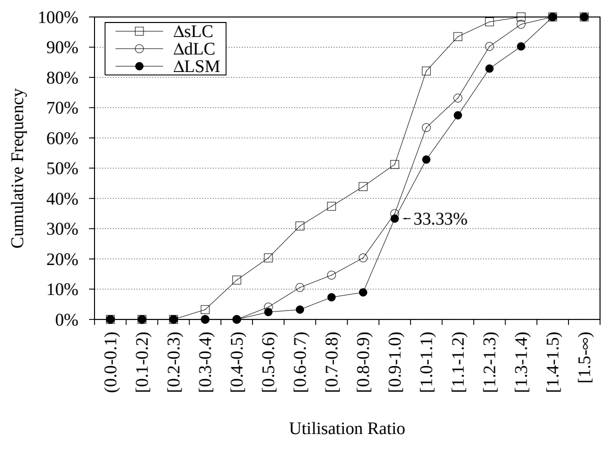

Figure 4. Cumulative frequency curves for the utilisation ratios

4.2 Performance evaluation

Although the effects of using the proposed memory hierarchy can be safely incorporated into the schedulability analysis, the performance advantages obtained from using the proposed memory hierarchy should be analysed.

Since a higher cache hit ratio does not necessarily guarantees that every task in the task set will satisfy its deadline, the approach used in this work to measure the quality of the solution is to use Processor Utilisation. The lower the processor utilisation, the better, since this means that the task set demands less CPU time and thus other tasks might be included in the task set while the system remains schedulable (i.e., all tasks executing on time).

Estimated Processor Utilisations for the system with an LSM (U_{LSMe}), the system with an LC used in a static manner (U_{sLCe}), and the system with an LC used in a dynamic manner (U_{dLCe}), were calculated by using the same GA for block selection and the appropriate I-cache refill penalty.

Afterwards, the different utilisations were normalised against the utilisation obtained when simulating the system with a conventional cache, U_{Cs}, to obtain the Utilisation Ratios ($\Delta U_X = U_{Xe}/U_{Cs}$, where $_X$ is one of $_{LSM}$, $_{dLC}$, $_{sLC}$).

Figure 4 shows that:

- In less than 34% of the cases, $\Delta U_{LSM} > 1$; i.e., $U_{LSMe} > U_{Cs}$, and hence, the proposed memory hierarchy brings about the same or better processor utilisation than that obtained when using a conventional cache in around 66% of the cases;

- In every range, $\Delta U_{LSM} < \Delta U_{dLC} < \Delta U_{sLC}$ and hence, the proposed memory hierarchy brings better processor utilisation than using a locking cache in a dynamic manner; moreover, in the zone with losses (range $[0, 1.0)$), the proposed memory hierarchy provides lower losses and in the zone with gains (range $(1.0, \infty)$), the proposed memory hierarchy provides higher gains.

Furthermore, a statistical analysis of three null hypothesis tests (t-test, sign test, and signed rank test) was done to corroborate that $\Delta U_{LSM} - \Delta U_{dLC} < 0$ (i.e., that the proposed memory hierarchy provides a better Processor Utilisation than the dynamic use of locking cache). The first one establishes whether the mean is zero or not; the remaining two tests allow to determine whether the median is zero or not. The sign test is based on counting the number of values above and below the hypothesized median, while the signed rank test is based on comparing the average ranks of values above and below the hypothesized median. All of the three tests revealed that $\Delta U_{LSM} < \Delta U_{dLC}$ at the 95% confidence level.

5 Concluding remarks

By virtue of including the LSM, any I-cache is transformed into a virtual locking I-cache, independently of its size, associativity and block size, the three main organisation parameters in a cache memory. In addition, parameters like I-cache replacement policy are irrelevant, provided that the algorithm used to select I-cache contents guarantees that there will be no conflict misses.

Results show that the proposed memory hierarchy is predictable and simple to analyse. Moreover, when compared to dynamic use of locking cache, it offers (i) a lower overestimation in predictability; and (ii) a higher performance. Finally, when compared to a conventional cache, in many cases the proposed memory hierarchy performs better or very close to it.

On the other hand, the proposed memory hierarchy does not needs explicit management of the memory hierarchy at run-time, while both scratchpad memories and locking cache memories, do. Moreover, the use of scratchpad memories requires explicit modifications in the application code's control flow.

In short, the memory assist is versatile in its operational aspects, yet it uses generic components; it does not cause any extra overheads to the system; its impact on system programming is negligible; and, it may be embedded in System-on-a-Programmable-Chip designs targeted to current FPGAs, while contributing in a significant way to determinism and performance improvements with respect to dynamic use of a locking I-cache.

All of these advantages are obtained at a fraction of the cost of the original system, thus paving the way to widespread use in realistic real-time systems.

References

[1] Alt M., Ferdinand C., Martin F., and Wilhelm R. Cache behavior prediction by abstract interpretation. *Lecture Notes in Computer Science (LNCS)*, 1145, Sept. 1996.

[2] Arnaud A. and Puaut I. Dynamic instruction cache locking in hard real-time Systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RTNS'06)*, pages 179–188, May 2006.

[3] Jacob B. L. and Bhattacharyya S. S. Real-time memory management: Compile-time techniques and run-time mechanisms that enable the use of caches in real-time systems. Technical report, Institute for Advanced Computer Studies, University of Maryland at College Park, USA, Sept. 2000.

[4] Jain P., Devadas S., Engels D. W., and Rudolph L. Software-assisted cache replacement mechanisms for embedded systems. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, Nov. 2001.

[5] Kirk D. B. SMART (Strategic Memory Allocation for Real-Time) cache design. In *Proc. of the 10th IEEE Real-Time Systems Symposium*, pages 229–237, Dec. 1989.

[6] Li Y.-T. S., Malik S., and Wolfe A. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 254–263, Dec. 1996.

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

[8] Martí Campoy A., Pérez Jiménez A., Perles Ivars A., and Busquets Mataix J. V. Using genetic algorithms in content selection for locking-caches. In *Proc. of the IASTED International Symposia Applied Informatics*, pages 271–276. Acta Press, Feb. 2001.

[9] Martí Campoy A., Puaut I., Perles Ivars A., and Busquets Mataix J. V. Cache contents selection for statically-locked instruction caches: an algorithm comparison. In *Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 49–56, July 2005.

[10] Martí Campoy A., Tamura E., Sáez S., Rodríguez F., and Busquets-Mataix J. V. On using locking caches in embedded real-time systems. In *Proc. of the 2nd International Conference on Embedded Software and Systems (ICESS-2005). Lecture Notes in Computer Science (LNCS) vol. 3820*, pages 150–159, Dec. 2005.

[11] G. McPherson. *Applying and Interpreting Statistics: A Comprehensive Guide*. Springer Texts in Statistics. Springer-Verlag New York, Inc., second edition, 2001.

[12] Mueller F. Compiler support for software-based cache partitioning. In *LCTES'95: Proc. of the ACM SIGPLAN 1995 workshop on Languages, Compilers, & Tools for real-time Systems*, pages 125–133, June 1995.

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

[14] Muller H., May D., Irwin J., and Page D. Novel caches for predictable computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, Oct. 1998.

[15] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, third edition, 2 Aug. 2004.

[16] Petrank E. and Rawitz D. The hardness of cache conscious data placement. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 101–112, 2002.

[17] Puaut I. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 217–226, July 2006.

[18] Puaut I. and Decotigny D. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, pages 114–123, Dec. 2002.

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

[20] Sebek F. Measuring cache related pre-emption delay on a multiprocessor real-time system. In *IEE/IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, Dec. 2001.

[21] Shaw A. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[22] Staschulat J., Schliecker S., and Ernst R. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 41–48, July 2005.

[23] Tamura E., Rodríguez F., Busquets-Mataix J. V., and 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 (WIP ECRTS'04). TR-UNL-CSE-2004-0010, Department of Computer Science and Engineering. University of Nebraska-Lincoln*, pages 1–4, June 2004.

[24] Tan Y. and Mooney V. A prioritized cache for multi-tasking real-time systems. In *Proc. of the 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'03)*, pages 168–175, Apr. 2003.

[25] Vera X., Lisper B., and Xue J. Data cache locking for higher program predictability. In *Proc. of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–282, June 2003.

[26] Wehmeyer L. and Marwedel P. Influence of onchip scratchpad memories on WCET prediction. In *Proc. of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 29–32, June 2004.

[27] Wehmeyer L. and Marwedel P. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)*, pages 600–605, Mar. 2005.

[28] Wolfe A. Software-based cache partitioning for real-time applications. In *Proc. of the 3rd Workshop on Responsive Computer Systems*, Sept. 1993.