Technical Report

A Context Aware Cache Controller to Bridge the Gap Between Theory and Practice in Real-Time Systems

Yannick Allard

Geoffrey Nelissen*

Joel Goossens

Dragomir Milojevic

*CISTER Research Center

CISTER-TR-140704

2014/08/20

A Context Aware Cache Controller to Bridge the Gap Between Theory and Practice in Real-Time Systems

Yannick Allard, Geoffrey Nelissen*, Joel Goossens, Dragomir Milojevic

*CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: grrpn@isep.ipp.pt, joel.goossens@ulb.ac.be

http://www.cister.isep.ipp.pt

Abstract

A Context Aware Cache Controller to Bridge the Gap Between Theory and Practice in Real-Time Systems

Yannick Allard*, Geoffrey Nelissen†, Joël Goossens*, Dragomir Milojevic*
*PARTS Research Centre
Université libre de Bruxelles (ULB)
{yallard, joel.goossens, dmilojev}@ulb.ac.be
†CISTER/INESC-TEC, ISEP
Polytechnic Institute of Porto
grrpn@isep.ipp.pt

***Abstract*—Nowadays, most processing platforms make use of cache memories to improve the execution speed of the tasks running on the processors. However, when a processor switches from a task to another, the caches must be reloaded with the context of the upcoming task. This is time consuming and is usually not predictable and thus affects the worst-case execution time of the task. Such unpredictability should be avoided in real-time systems in which the instant at which a result is available is as important as the result itself.**

In this paper, we present a hardware component named *hardware context switch* (HwCS) which replaces the standard L1 cache controller a processor. It divides the cache in two interchangeable layers and enables to save or restore the content of one layer while the second is *simultaneously* used as a usual cache by the processor. Saving the cache content after a preemption and restoring this content before resuming the execution of the preempted task, makes the preemption overheads negligible in comparison to the task worst-case execution times. It is theoretically proven that the existing scheduling theory can be used "as is" with the HwCS by simply reducing the task deadlines, thereby bridging the gap between theory and practice. The HwCS has been implemented in an uniprocessor system as a proof of concept. The first results show a neat improvements on the processor utilisation for a small cost in silicon surface.

I. Introduction

Over the last few decades, the technology used in integrated circuits has been tremendously improved. Both the speed and the number of integrated logic gates double each 18 month, this trend is commonly known as the Moore's Law. The high number of transistors that are now available in a same die, makes possible the design of multi- and many-core systems composed of more than 250 cores [1]. However, memory technology scaling and thus read/write access time did not follow the same trend as the processor logic scaling. To circumvent this problem modern processors usually use a *hierarchical memory* architecture. With such architecture, small and fast local SRAM memories called *caches* are placed near the processors to alleviate the latencies of a slow central DRAM memory. Whenever instructions or data are accessed by a task running on a processor, they are saved in the local cache of this processor. Hence, the subsequent time that the task will access those instructions (or data), they will be available in the fast local cache and the task will not have to make an access to the slow central memory.

Because a cache is smaller than the main memory by several order of magnitude, data have to be organised to be quickly accessible: a cache is used to keep track of temporal and spatial locality of data or instructions accessed by a task running on a processor. When a task running on a processor reads data that is not stored in the cache, there is a *cache miss*. Because data must be read in a memory located further from the processor, the time to access the requested instruction or data will be significantly higher — about 10 or 100 times longer [2] — than the time needed to access data already stored in the cache (*i.e.*, if there is a *cache hit*).

In practice, the memory architecture may be composed of several *levels* of caches starting from the *level 1 cache* (L1) which is closest to the processor and has fastest access time going to up to the L3 cache which is usually the last level before the system main DRAM memory.

Unfortunately, this hierarchical memory architecture decreases the predictability of the task execution times. Let us for instance consider the case of a task τ_a running in *isolation*, *i.e.* running alone on the platform. In this situation, τ_a has a given worst-case execution time (WCET) C_a. Now, let us assume that τ_a is scheduled with other tasks on the platform and another task — say τ_b — preempts τ_a during its execution on a processor π_j. After the completion of τ_b, the task τ_a can be resumed on processor π_j. However, because τ_b executed on π_j, instructions and data of τ_a that were stored in the L1 cache have been (at least partially) replaced by the instructions and data of τ_b. Thus, when resumed, the task τ_a must look for its instructions and data in higher levels of caches or even in the central memory. Because the WCET of τ_a might depend on other tasks running in the system, the WCET might be longer than the one determined in isolation due to the increased number of cache misses, studies proposed new methods to compute a more accurate WCET [3].

All these preemption overheads must be added to the initial worst-case execution times of the tasks [4]. These overheads may be really high as shown through an empirical study in [5] where preemptions delays may reach up to 1 million processor cycles on a platform providing three levels of caches.

This problem should even worsen with an always increasing number of processors available in processing platforms. The communication delays between the processors and the central memory will indeed increase with the number of processors trying to access the memory through the communication network, thereby increasing the number of contentions on both the communication and memory levels and inducing a significant jitter on the access time.

This jitter on the worst-case execution time of the tasks is unacceptable in the context of a hard or firm real-time system. Indeed, the tasks have deadlines that must imperatively be respected (at least with some predefined quality of service in the case of firm real-time systems). The worst-case execution times of the tasks have therefore to be known *a priori* to perform schedulability tests, which will be able to say whether the task set is schedulable or not.

The impact however depends on (i) the specific scheduling algorithm (and its implementation) utilized by the operating system to schedule the tasks [6], [7] and (ii) the architecture of the hardware platform [6].

There are several ways to address this problem. First, we can try to integrate the preemption costs in the theory analyzing the schedulability of a system. This approach however, usually entails a large degree of pessimism [4], [5] and a task set which is actually schedulable may end up being deemed unschedulable by the modified schedulability test [7]. The second approach consists in either using properties of the architecture or integrating new hardware services in the platform to improve the system predictability and reduce overheads [8]–[11], usually these services require processor modifications. Finally, designing a specific architecture with high predictability and specific design flow is also an option [12]. We take the second option in this paper. Specifically, we present a hardware component — named *hardware context switch (HwCS)* — which is based on a simple variant of a cache memory controller. The HwCS is build upon the concept of double buffering applied to the cache state (data and data organisation). It is designed to replace the standard L1 cache controllers in the platform architecture and is therefore almost independent from the processor used in the platform. The HwCS makes the preemptions negligible for the executed application, thereby making the preemption costs virtually equal to zero for the tasks. As later proved in Section IV, this property has for strong consequence that all the real-time scheduling theory can now be used *"as is"* by only adapting the deadline associated with each task, as long as the schedulability tests are still correct with the modified deadline.

The HwCS achieves this important result by providing multiple *layers* of caches. Each cache layer has capabilities to either save its content in the main memory or load a previously saved content from the main memory, while another layer is used as a usual L1 cache by the processor. The simultaneity of those operations enables a processor π_j to continue to execute a task while the complete execution context of the next task to be executed on π_j is loaded in a second cache layer. Once loaded, the next task can start its execution on π_j in the exact same state as it was left when it was preempted. Therefore, the number of cache misses and hence the worst-case execution time of the tasks will not be affected by the preemptions anymore.

The proposed approach of multi-layered cache is very well adapted to recent evolutions in integrated circuit (IC) packaging, namely 3D-Stacking. In such configuration all system SRAMs could be implemented in one or more independent circuits placed on the top of the logic die. Each cache layer could be implemented with the same memory size as in the 2D counterpart, but without increase of the overall IC footprint and loss in memory access performances due to physical distance of the memory from the processor.

Although the HwCS can be used to enhance migrations on multiprocessor platforms, this paper will focus on preemptions on uniprocessor systems.

Organisation of this paper: Related works are first presented in Section II. The concepts, principles and correctness of the HwCS are then explained in Section III. Section IV studies the impact of the HwCS on the real-time scheduling theory. Experimental results are provided in Section V. Section VI concludes the paper.

II. Related works

Low-end microcontrollers have often a direct access to the main memory [13] enabling access time full predictability. However, although this can be implemented for small systems with a few hundred kilobytes of memory, it is not practical with megabytes or gigabytes of memory we can find in nowadays systems.

In this context, cache memories have been used for decades [14] but predictability is lower than systems with direct memory access because data may or may not be in cache, implying a time consuming reload which worsens the WCET [5].

Other hardware services allow fast preemptions. We propose a review of them in this section before reviewing studies about impact of preemptions on the WCET.

A. Hardware services for fast preemptions

1) Processors with multiple register banks: These processors take advantage of several hardware register banks to switch between contexts. Commercial processors have usually 2 register banks. There are several variants around this concept. For instance, SPARC processors take advantage of up to 32 overlapping register windows to switch context and pass arguments/results between contexts [15]. Some industry RISC processors like ARM$^{®}$processors, SHARC$^{®}$or academic processor like COFFEE use two almost independent register files to switch context in a few clock cycles [16]–[18]. We can cite the NEC-DRP1 experimental processor which has 16 register banks [19]. This architecture shows improvements of 1.6-6x in execution speed compared to usual architectures [19]. Unfortunately, this approach limits the number of contexts stored in a processor to the number of register banks. If more contexts than the number of register banks are necessary, then they have to be stored outside the processor when not in use and reloaded on the processor when needed as on any single register bank processor. Moreover, it does not solve the increase of the task execution time due to the higher number of cache misses caused by the execution of multiple tasks on the same processor.

Multithreaded processors [20], [21] is another example of architecture multiplying the number of register banks. Fine-grain multithreading is used in the Tera computer architecture [20] using a barrel-processor with one register set per thread state. Coarse-grain multithreading is used in consumer oriented processor [21] and is based on context switching between two states of the processor when a stall happens. Again, the processor state must be duplicated. The L1 cache is shared between the threads.

2) Specific virtual platform mapped to hardware: It is possible to virtually allocate resources of a cacheless system to tasks and implement a time division multiplexing (TDM) scheme to bound usage of resources and so ensure real-time behaviour with predictability, like the CompSOC [12] architecture. This platform relies on partitioning and virtualisation of applications: all resources (processors, memories) are budgeted and scheduled to ensure full predictability. Caches and interrupts are unavailable on this platform. This platform has its own design flow and its own architecture, different from usual systems.

3) Partitioned and locked cache memories: Some cache implementations add "cache line locking" and/or "cache partitioning". This ensures that some lines stay in the cache whatever happens to other data in the cache. This can lead to predictable replacement behaviours in the cache [2] but prevents a part of the cache to be useful when locked lines are present in the cache. Nevertheless, cache partitions must still be shared by multiple tasks if there are fewer partitions than tasks. Moreover, the more partitions there are, the smaller the actual size of the L1 cache available for a task is. To some extents, the same goal can be reached with scratchpad memories where software can decide which data is in the memory at which time. Note however that some studies showed that cache locking can improve the worst-case execution time of the tasks [22]. There is no way to save the state of a locked zone of the cache to free some space and to reload it later to resume execution in the same state.

4) Specific hardware services: Integrating new hardware services in a platform to improve the system predictability and reduce overheads is also possible [8]–[11]. These services show good results at reducing overheads and are often added at the processor level. The use of an ID for each process can help reconfigure the memory management system to speed-up context switching. Unfortunately, the number of processes is limited to 128 in the ARMv6 architecture [10], [16]. Another approach is to modify the processor to add save/restore abilities for more than one register set [23]. In the Rhamma processor, the register sets stored near the processor for switching can also be copied to the memory, thus removing the limit on the context number. We will use a similar technique but at the L1 level.

B. Preemption overheads impact on the WCET

Most of the theoretical works on the real-time scheduling assume that the worst-case execution time C_i of any task τ_i includes all the potential overheads this task could suffer during its execution, especially preemptions. However, such an approach can be very pessimistic since the number of preemptions that a job may incur in the worst-case can be far more larger than the number of preemptions it suffers in the average and hence most frequent case. As a consequence, the processing platform is over-dimensioned and lots of computational resources are left unused because of the overestimation of the task computational needs.

Some works [24], [25] account for the overheads caused by each task τ_i rather than the overheads suffered by τ_i. This approach helps to reduce the pessimism of the overheads accounting for some scheduling algorithms such as fixed job priority algorithms.

Unfortunately, the preemption overheads may impact the execution time of a task more than once, thereby highly increasing its WCET. Furthermore, deriving an exact upper-bound on the number of preemptions that a job can either suffer or cause can turn out to be extremely difficult for some scheduling algorithms with dynamic job priorities. Consequently, there exist scheduling algorithms for which neither of those approaches has been successfully applied yet [26] or for which the computed worst-case execution time becomes overly pessimistic [27]. Therefore their schedulability tests remain theoretical and their performances for the scheduling of realistic systems cannot be estimated differently than by conducting empirical studies.

III. Hardware Context Switch (HwCS)

To reduce the impact of preemptions and migrations on the execution time of tasks, we designed a HwCS which almost removes the burden preemption from the processor and transfer it to a context aware L1 cache controller.

We assume a generic uniprocessor architecture with a L1 cache and potentially other caches in the memory hierarchy. Von Neumann and (modified) Harvard architecture are both supported.

A. Concepts

A cache memory keeps track of spatial and temporal locality of accesses made to a memory so that the cache memory can speed up the future accesses to pre-stored data. Each cache line holds (i) data, (ii) a tag which is used to store the base address of data in the main memory and (iii) a few flags (valid line, modified line) used by the cache controller. Sometimes, data stored in the cache must be replaced with other data. The choice of the line to be replaced in the cache is done by the cache controller. Lots of replacement policies have been developed over the years and the choice can be based on FIFO, random, least/most recently used line [14], [28] or any other policy [29], [30].

As explained in the introduction, the preemption of a task τ_a by a second task τ_b will trigger a replacement of the cache content with the instructions and data of τ_b. Therefore, a part (if not all) the execution context of τ_a will have been lost when τ_a will be resumed. A lot of cache misses will therefore occur and the execution of τ_a will be slowed down. Without the HwCS, after a context switch, a processor usually waits for the cache controller to reload its content and so can be stalled for a long time waiting for the cache misses to be resolved.

A good technique to avoid these cache misses related to the context switches would be to save the execution context of τ_a in the main memory in order to restore it later in the cache of the processor. Hence, whenever a job is resumed after a preemption, the temporal locality and the spatial locality of data present in the cache is the same as before the preemption. The burden to reload the cache memory after a preemption is now supported by the HwCS and does not affect the processor anymore because another layer can be used as a usual cache at the same time for the current task, thereby limiting the preemption overheads to only few clock cycles needed to save and restore the internal processor context (i.e., the values of the internal registers of the processor) and to the time needed to configure the HwCS operations. Furthermore, the number of tasks available for preemption is limited by the reserved space in the system memory to store layer states.

The basic idea of the hardware context switch is the following: the private L1 cache of a processor is divided into two or more *layers* (see Figure 1(b)) of the same size. The processor can access only one layer at a time, thereby implying that the size of the cache visible by the processor is equal to the size of one layer. Moreover, the layer which is not currently used by the processor can be in three different states: (i) idle, (ii) saving its content in the main memory or (iii) restoring a previously saved content from the main memory.

The different layers are not used to ease keeping spatial and temporal localities in separate caches [31] to improve performances like in dual-cache systems. In our design, all layers have the same properties regarding spatial and temporal localisation of data.

This proposed new HwCS enables to *simultaneously* execute a task on the processor and restore the execution context of the next task to execute or save the previous executed context in the other cache layer. Three situations may therefore occur:

1) The processor is executing a task — say τ_a — using the first layer of the HwCS as a usual L1 cache (see Figure 2(a)). The second layer of HwCS remains idle.
2) The scheduler decides to preempt the execution of τ_a to start executing another task τ_b instead. The HwCS therefore starts to restore τ_b's context in its second layer while the processor continues to execute τ_a using the first layer as a usual cache memory (see Figure 2(b)). In this case, the processor is not even aware that τ_b is being prepared for execution.
3) The execution context of τ_b has been completely restored in the second HwCS layer. The processor hence starts running τ_b using this second layer as its L1 cache (see Figure 2(c)). In parallel, the content of the layer which is related to the execution of τ_a is saved in the main memory.

This idea is quite similar to a double buffering scheme applied to the whole L1 content including data, tags, flags and replacement information.

The only unavoidable operations performed by the processor are: (i) choosing which HwCS layer must be used for the execution of the current running task, and (ii) configuring the layer controller to initiate the saving or the restoring of the cache content.

If the size of a layer is smaller than the size of the L1 original cache, the task using it will suffer more cache misses [32] during its execution but preemptions will no longer be impacted and will cost virtually zero CPU time.

To the best of our knowledge, dual-buffering the whole L1 (data, tags, flags and replacement information) and adding abilities to save and restore a layer simultaneously has never been implemented to improve system predictability.

B. Saving and restoring policy

1) Solution 1: The easiest implementation of the HwCS would consist in saving the whole cache layer content including tags, flags, data and the replacement policy information, at consecutive addresses in the central memory. When this cache state should be restored, the HwCS will simply have to read the layer content as one block in the main memory.

Although the additional complexity of the cache controller is minimal with this method, it faces important disadvantages:

a) Systematically saving the data and instructions stored in the cache layer increases the amount of data transferred through the communication network.

b) Saving the data of the cache will cause data duplication and pollution of L2/L3 caches. The Figure 3a shows this process. Let X be a piece of data stored in the cache layer used by a task τ_a. X is accessed at the address $\text{addr}(X)$ by τ_a (accesses marked 2'). However, when the state of the cache layer used by τ_a is saved in the main memory during a context switch, the value of X is saved with the rest of the cache content at another address denoted by $\text{addr}_{\text{save}}(X)$ (accesses marked 1'), so X exists now at two different addresses. Hence, when the execution context of τ_a should be restored in a cache layer, the HwCS will read the value of X at the address $\text{addr}_{\text{save}}(X)$ thereby warming up the L2 and L3 caches with tags related to $\text{addr}_{\text{save}}(X)$ (accesses marked 1) and not to the address $\text{addr}(X)$ at which τ_a accesses X (accesses marked 2). This may eventually slow down the execution of τ_a because a part of the context of τ_a that was still in L2 and/or L3 caches might be evicted and will be reloaded at the first subsequent access to X.

c) Coherency with data shared with other tasks is not guaranteed. Assuming X is shared with a second task τ_b. The value of X might be modified by τ_b when τ_a is not running on the platform. Those modifications will be saved at the address $\text{addr}(X)$ in the main memory. However, because a copy of X was saved at the address $\text{addr}_{\text{save}}(X)$ when τ_a was preempted and because the execution context of τ_a will be restored using $\text{addr}_{\text{save}}(X)$, the modifications made to X will not be seen by τ_a. There is therefore a problem of coherency on the value of X between the tasks.

2) Solution 2: Most of these problems can be solved by not saving a copy of the data and instructions stored in each

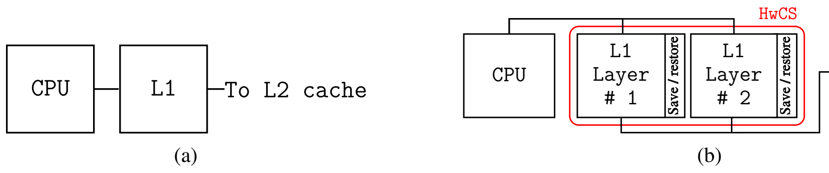

Fig. 1. (a) Simple Von Neumann or Harvard modified memory architecture. (b) Simple Von Neumann or Harvard modified memory architecture with a 2-layer HwCS.

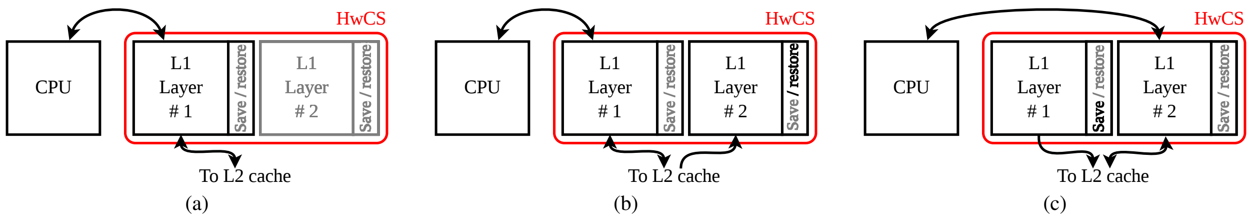

Fig. 2. Three possible scenarios with the HwCS. (a) The first layer is used as a cache while the second is idle. (b) The first layer is used as a cache while the second restores the context of the next task to be executed. (c) The first layer saves its context in the main memory while the second is used as a cache.

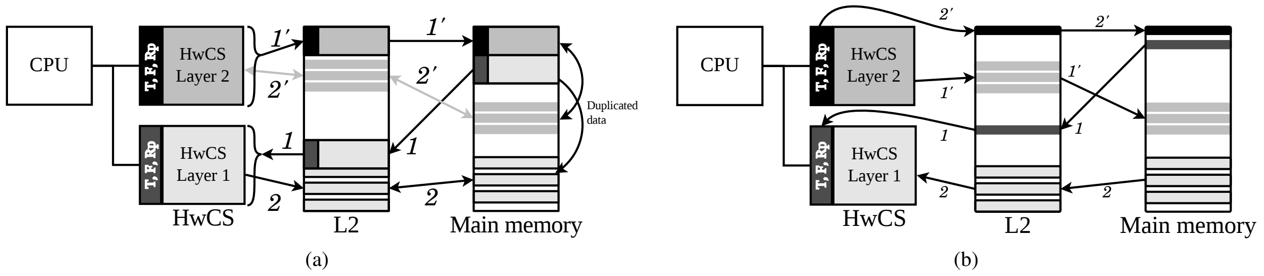

Fig. 3. Comparison of solutions 1 and 2. **Solution 1 (a)**: Save operation (top): Data, Tag, Flags and replacement policy information are saved to the main memory (1') and cause allocation in L2. Data already in L2 (2') are duplicated with a different locality. Restore operation (bottom): Data, Tag, Flags, replacement policy information are restored from the main memory (1) and cause allocation in L2 with data related to the context address in main memory. These data are not used later by the task. And any write made in the HwCS, if propagated to the L2, will cause reallocation in L2. **Solution 2 (b)**: Save operation (top): data are first updated in L2/main memory (1') then Tags, Flags and replacement policy information are saved to the main memory causing minimal allocation in L2 (2). Restore operation (bottom): Tag, Flags, replacement policy information are restored from the main memory first (1) and cause minimal allocation in L2. Then data are refreshed in the HwCS (2), causing L2 warm up with data actually used by the task.

cache line along with the tags, flags and the replacement policy information. Indeed, the tags contain the addresses of the data and instructions previously stored in each cache line. With that, the cache data memories can be restored or saved without data duplication. Any modified line must be updated to the main memory during a layer save as shown on Figure 3b, phase 1', and then the tags and flags have to be saved in the main memory, phase 2.

For a restore operation, the tags and flags must be restored first, phase 1, and then all lines must read from the main memory, phase 2. Using these methods, data shared by tasks are up-to-date and data are not duplicated, thereby removing the coherency issue.

Restoring data using the tag causes transfers up to the size of the cache and triggers allocations (warm-up) of data in the L2/L3 caches *i.e.*, the data stored in the layer are also stored in L2/L3 caches, so cache misses related to the resumed task might be resolved sooner.

With the solution 1, when we save the cache layer state, the transfer size with the main memory is the sum of the tag and flags, data and the replacement policy information memories. Whereas with the solution 2, we only store the tag and flags and the replacement policy information memories. The data are updated in the main memory during the save operation and reloaded during the restore operation, implying additional transfers up to the size of the layer. For example, this transfer is 5120 Bytes for a 32 kB 4-way associative cache using 32-Byte lines with a FIFO replacement policy, which represents only 16% increase related to the cache size. Solution 2 will use only 5120 bytes for each context whereas Solution 1 will use 37888 bytes).

C. Restrictions

1) Two Layers configuration: In the architecture described above one layer of the HwCS is used by the processor as a usual cache while the second layer may be used to save or restore another task context. As pictured on Figure 4, when a task τ_b must preempt a task τ_a, it first restores the context of τ_b in the second layer (represented by a rising ramp on Figure 4) while τ_a is still executed on the processor. Then, when the execution context of τ_b has been fully restored in

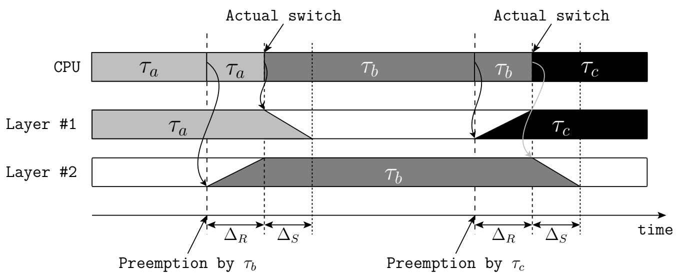

Fig. 4. Context switching with two layers

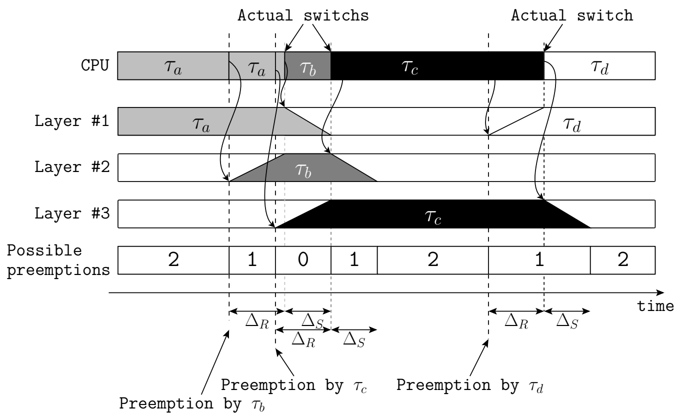

Fig. 5. Multiple preemptions with 3 layers

the second cache layer, τ_b starts to execute on the processor and the execution context of τ_a can be saved in the main memory (represented by a falling ramp). The save and restore operations therefore happen in a row whenever a context switch occurs.

Restoring or saving a layer takes some time as it has to be read or written in the main memory respectively. This time is dependent on the memory bandwidth and can usually be bounded. The upper-bound on the the time needed to restore the state of a cache layer is denoted by Δ_R. Similarly, Δ_S provides an upper-bound on the time required to save the context of a cache layer in the main memory. As we can see on Figure 4, in the worst-case, no layer is available during $\Delta_R + \Delta_S$ time units to restore the context of a new task. This imply that no new preemption can happen during this interval. Therefore, the minimum time between two consecutive preemptions on a same processor is equal to: $T^{\min}_{\text{preempt}}(N_{\text{layers}=2}) = \Delta_R + \Delta_S$

2) More than Two layers configuration: This minimum time imposed between two consecutive preemptions can be reduced by increasing the number of layers in the HwCS. Indeed, restoring and saving task execution contexts can be processed in parallel on different layers. For instance, as shown on Figure 5, if we have three layers in the HwCS, the execution contexts of two different tasks can be restored in an interval of $\Delta_R + \Delta_S$ time units. More generally, we can restore the execution contexts of $N_{\text{layers}}-1$ tasks in an interval of $\Delta_R+\Delta_S$ time units when there are N_{layers} layers in the HwCS. The time between two simultaneous job arrival can be very short — virtually zero — as long as at least a layer is empty, so $T^{\min}_{\text{preempt}}(N_{\text{layers}>2}) = \varepsilon$.
A practical limitation may arise from the required bandwidth necessary to save/restore contexts. The number of concurrent transfers with the memory increases with the number of layers as more than one save/restore operation occurs at the same time. Consequently, the delays Δ_R and Δ_S can be longer than in a baseline design if the actual memory bandwidth is kept constant.

D. Performance impact

The unpredictability of the cache related to the context switching is no longer a concern as the processor does not suffer additional cache misses caused by preemptions; the cache state is restored as it was before the preemption. Cache misses due to context switching are therefore avoided.

Although the HwCS enables the processor to seamlessly switch between tasks without major overhead, this ability has two main drawbacks:

- Each layer of the HwCS must be able to save its state or restore its state from the main memory. This increases the controller complexity and might impose to provide one cache controller for each layer.
- The bit-rate on the communication network as well as the number of memory requests is increased since the save and restore operations add additional transfers on the architecture.

These additional costs imposed by the HwCS have been measured and the results will be presented in Section V. However, in the next section, we first study the impact of the HwCS on the real-time scheduling theory.

IV. Bridging the gap between theory and practice

In the previous section, we described the principles and concepts of the hardware context switch. We said that the uniprocessor platform should have its L1 cache replaced by such a hardware context switch so that the predictability of the platform would be improved. In this section, we show the implications of the hardware context switch on the application behaviours running upon the platform and indeed prove that the predictability of the hardware context switch makes the practice fit the theory.

When we say that a task set is *theoretically schedulable* with a given scheduling algorithm $\mathbb{S}$, we mean that all the job deadlines would be respected by $\mathbb{S}$.

In the following, we will use the terms *HW schedule* and *OS schedule* to differentiate between the schedule actually executed on the hardware platform (*i.e.*, the exact instants at which the tasks start to execute on a processor as well as the exact execution times of those tasks) and the schedule determined online by the OS scheduler assuming that all the decisions taken online by the operating system were instantaneously realized (*i.e.*, if the preemptions had no cost and there was not any delay imposed by the HwCS), respectively. Note that the HW schedule directly depends on the OS schedule computed online but is impacted by the actual hardware characteristics.

Let us assume the *initial task set* $S_\mathrm{I} \overset{\mathrm{def}}{=} \{\tau_1, \ldots, \tau_n\}$ of sporadic tasks. Each sporadic task is defined as $\tau_i \overset{\mathrm{def}}{=} \langle C_i, D_i, T_i \rangle$ *i.e.*, characterized by a worst-case execution time C_i, a deadline D_i and a minimum inter-arrival time T_i. That is, the task τ_i releases a potentially infinite amount of jobs. The release of two jobs are at least T_i time units apart, and each of those job must execute for at most C_i time units before its relative deadline occurring D_i time units after its release. If a job ends earlier than its WCET, the processor will stay idle until the next decision taken by the scheduler is applied.

We will focus on the demonstration for uniprocessor designs. Multiprocessors designs and migrations are left for future works.

In this section, we make the following assumptions:

Assumptions ***1–4***

1) *The operating system (OS) makes its scheduling decisions relying on the* OS schedule. *That is, the operating system assumes that no job ever exceeds its worst-case execution time and that the tasks start executing on the platform right at the instant at which the OS made the decision (even though it does not correspond to what actually happens on the platform);*
2) *At most, $N_{layers} - 1$ preemptions happens in any interval of $\Delta_R + \Delta_S$ time units when there are N_{layers} layers in the HwCS;*
3) *Loading the context of a task to be executed on the processor always takes exactly Δ_R time units (including for the first execution of the task, i.e., when there is no context to restore yet);*
4) *If the tasks share resources, the resource critical section is executed non-preemptively.*

Assumption 2 comes from the limitations of the HwCS explained in Section III-C. Assumption 3 means that if the save/restore operation is shorter than Δ_R, the processor will wait until Δ_R time unit are elapsed before switching.

The last assumption (Assumption 4) ensures that a task never stops to execute on a processor because of an unaccessible resource and hence the time spent to access resources is included in the worst-case execution times of the tasks. The use of other resource sharing protocols such as "Priority Ceiling" or "Priority Inheritance" will be investigated in future works.

A. Impact of the hardware context switch on the theory

The hardware context switch presented in the previous section, makes the preemption cost negligible when compared to the worst-case execution time of a task (see Section V for accurate values). Furthermore, because the state of the cache memory is exactly the same when a task τ_i restarts its execution as it was when τ_i was preempted, τ_i's execution is not slowed down by the preemption. Therefore, the worst-case execution times of the tasks are barely impacted by the preemptions when the hardware context switch is used, and in a first reasonable estimation, the worst-case execution times can be considered completely independent of those.

However, because the hardware context switch must load the cache with the execution context of a task τ_i before starting to execute τ_i, a *delay* appears between the exact instant at which τ_i starts running on a processor and the instant initially scheduled for its execution by the OS scheduling algorithm. We now analyse the impact of this delay on the timing behaviours of the system.

Lemma 1. *The HwCS will delay any scheduling decision taken by the* OS scheduler *by Δ_R time unit in the* HW schedule *related to the OS schedule.*

Proof: This is the direct consequence of Assumptions 2 and 3. ■

A consequence of Lemma 1 is that nothing will be executed on the processor during Δ_R time unit after the system start. Therefore, the execution time of all the tasks in the HW schedule is always equal to the execution time computed by the scheduling algorithm for the OS schedule. Furthermore, the HW schedule is identical to the OS schedule shifted forward by Δ_R time units (see Figure 6 for an example of the result of an online schedule). Hence, the preemptions have no impact on the actual execution times of the tasks.

However, because the OS schedule is shifted forward by Δ_R time units, if a job was supposed to end its execution right at its deadline in the OS schedule, it will end up finishing its execution Δ_R time units after its deadline in the HW schedule (see Figure 6). Therefore, to be certain that all jobs will respect their deadlines in the HW schedule, we must reduce their deadlines by Δ_R time units in the OS schedule and hence impose a virtual deadline $D_i^{\mathrm{OS}} \overset{\mathrm{def}}{=} D_i - \Delta_R$ to every task τ_i.

Let us assume the modified task set $S_\mathrm{OS} \overset{\mathrm{def}}{=} \left\{\tau_i^{\mathrm{OS}} \overset{\mathrm{def}}{=} \langle C_i, D_i^{\mathrm{OS}}, T_i \rangle \mid \tau_i \in S_\mathrm{I}\right\}$, the following theorem holds:

Theorem 1. *If the modified task set S_OS is (i) theoretically schedulable by a scheduling algorithm $\mathbb{S}$ and (ii) Assumptions 1 – 4 are respected, then S_I is schedulable when executed on a platform using the hardware context switch.*

Proof: By the properties of the hardware context switch, the worst-case execution times of all tasks in S_I are not impacted by the preemptions. Therefore, the worst-case execution time experienced by any task τ_i^{OS} in the HW schedule executed on the platform can be considered as being equal to its worst-case execution time in the OS schedule.

Let $a_{i,j}$ be the time when the j^{th} job of task $\tau_i \in S_\mathrm{I}$ (and thus of task $\tau_i^{\mathrm{OS}} \in S_\mathrm{OS}$) is released, and let $f_{i,j}^{\mathrm{OS}}$ and $f_{i,j}^{\mathrm{HW}}$ be the times when the j^{th} job of task $\tau_i^{\mathrm{OS}} \in S_\mathrm{OS}$ completes its execution in the OS and HW schedule, respectively.

Because the task set S_OS is theoretically schedulable by $\mathbb{S}$, all the jobs released by every task $\tau_i^{\mathrm{OS}} \in S_\mathrm{OS}$ will respect their virtual deadlines D_i^{OS} and therefore complete their executions at most D_i^{OS} time units after their release in the OS schedule

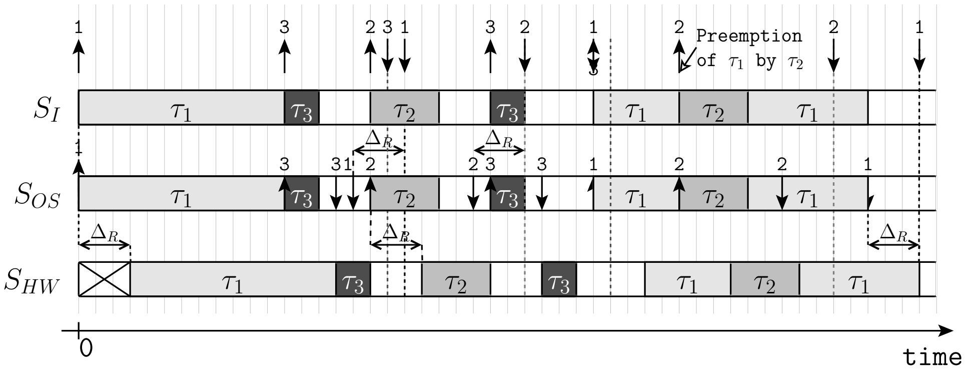

Fig. 6. Example schedules for initial, OS and HW task sets. Deadlines in the OS schedule are Δ_R shorter than in the initial task set. The HW schedule is the OS schedule shifted by Δ_R. With the reduced deadlines, all deadlines of the initial task set are met by the HW schedule. The crossed time units at the beginning of the HW schedule are unavailable for computation. All these schedules are what would result of the OS schedule computed online.

computed by $\mathbb{S}$. Formaly, we have $\forall i, j,\ f^{\text{OS}}_{i,j} \le a_{i,j} + D^{\text{OS}}_i$. Moreover, by Lemma 1, the hardware context switch will delay the execution of all jobs (both start and completion time) for all tasks in S_{OS} by Δ_R time units in the HW schedule executed on the platform. Since the WCET in S_{OS} and in S_{I} are the same, the job executed in the hardware schedule will finish at $f^{\text{HW}}_{i,j} = f^{\text{OS}}_{i,j} + \Delta_R$. And because $f^{\text{OS}}_{i,j} \le a_{i,j} + D^{\text{OS}}_i = a_{i,j} + D_i - \Delta_R$, there is $f^{\text{HW}}_{i,j} \le a_{i,j} + D_i$ thereby implying that all jobs of any task $\tau_i \in S_{\text{I}}$ respect their deadlines.

■

This is an important progress for the real-time scheduling community since scheduling algorithms can now be compared on a fair basis. For instance, an optimal scheduling algorithm is now indeed optimal, whatever the number of preemptions it may cause. When doing comparisons, a scheduling algorithm should now be considered better than an other one on the basis of more relevant properties such as the model of tasks it can schedule, the complexity of the algorithm, its behaviour in the presence of a given set of events, etc.

V. Experimental results

A. Experimental setup

The experimental platform used to test the HwCS was designed using a modified Harvard architecture with a HwCS composed of two layers. Specifically, a single ARM®Cortex™M0 processor with an AMBA AXI4™wrapper is wired to a 2-layer HwCS for combined data and instructions (2×4kB, 4-way associativity, 32-byte lines). The system has also one memory of 8MB (off-chip SSRAM with an AXI wrapper) to store the OS, tasks, data, instructions and the layer states.

We designed this setup to be predictable and easy to implement. Therefore, the three following choices have been made:

(i) the memory bandwidth is equally divided between all the layers; This is implemented using Time Division Multiple Access (TDMA) .
(ii) the save/restore operations transfer the whole content of the cache memory (data and instructions stored in the cache) along with tags, flags and the replacement policy information using the solution 1.
(iii) the baseline configuration keeps the same memory bandwidth as one layer of the HwCS (thus using only half the available bandwidth).

The setup is similar to the one presented Figure 1b with the L2 memory replaced by the SSRAM.

In this section, we will present the gate level implementation of a two-layer HwCS and we will compare the results to the baseline cache design. Then we will show the context switch performance of the HwCS.

B. Gate level implementation

The HwCS was implemented in VHDL starting from a standard cache controller design [14] using a FIFO replacement policy. We added necessary logic and data paths to the initial design to save and restore the cache tags, flags and replacement policy information. An address range is reserved to receive commands (save/restore/flush) from the CPU side. This cache controller uses three RAM to store data, tag and flags and the FIFO replacement policy information.

RTL synthesis were made using Synopsys Synplify Premier (E.2011-3).

The experimental platform enables the HwCS to be also configured as a standard baseline cache by removing the added features, thus using the same cache controller behaviour in both configurations. The area of baseline and two-layer HwCS are reported in Table I for different cache/layer size.

A HwCS layer is 10% bigger than the baseline cache controller, considering Flip-Flops (FF) and Look Up Tables (LUT) used for logic. Because we use a two-layer configuration, the logic used is multiplied by 2.2 and the RAM is doubled for the same HwCS layer size.

If we split the baseline cache size into two layers, we keep the same increase in logic area with a constant memory use.

Related to the whole platform, the increase in area is lower. If we consider the whole system with CPU, HwCS and glue logic compared to the same system with a baseline cache, the increase in area is 35% for logic and 100% for memory for a constant HwCS layer/cache size and only 35% for logic when splitting the baseline cache size in two for the HwCS.

The area of the Cortex M0 we use is around 950 FF and 3000 LUT which is similar to the logic area of one layer of the HwCS, so the increase seems important, but if we compare with a bigger processor, the increase in area is quite low.

For instance, according to [33], [34], the die of an Itanium 2 processor has a size of 421 mm^2 and the two 16kB L1 area used to store data and instructions have a total approximative area of 11.5 mm^2 on a 0.18 μm process. The micrograph of the die shows that approximatively half of the L1 die area is reserved for storage and the other half is logic. When scaled to this particular technology node and application, the area requirements for the HwCS with 2 layers are not more than 12.7 mm^2 for the cache controllers (recall that we now need two controllers, one for each layer) and 11.5 mm^2 for the RAM area needed for both layers (assuming that each layer has a size of 16kB). Consequently, the modified die would size 433.7 mm^2 which is only a 3% increase when we replace the L1 cache by a HwCS composed of two layers.

C. System level experimental setup and characterisation

The system clock runs at 70 MHz in a Virtex5 LX330 FPGA circuit on a CHIPIT®PLATINUM FPGA prototyping platform from Synopsys®. All measurements were performed using the gate level simulation of the platform.

A custom RTOS was developed to use the HwCS and apply the process shown on Figure 4 using fully independent tasks. It can be configured to run with a baseline cache as well. Whenever a context switch should occur on the processor, two interrupts are arisen. The first one executes the scheduler and configures the HwCS to start the restore operation of the next task to be executed. The second happens Δ_R later, switches contexts, configures the HwCS to save the other preempted context and starts the execution of the new task. Those interrupts represents the OS overhead listed in Table I. If theses interruptions are not cached, their worst case execution time is $200\mu s$ with the HwCS and $190\mu s$ on the baseline system.

1) Platform characterisation: We characterised our platform and measured the time needed to perform a cache flush in the worst case (all the cache/layer has modified data which must be updated in the main memory), the worst case time needed to reload all lines from the main memory, the sum gives the worst time needed to perform a context switch with the baseline cache: $T_{\text{min}}^{\text{preempt}}$. The OS overhead is also listed. Knowing $T_{\text{min}}^{\text{preempt}}$, the worst case available time (WCAT), *i.e.*, the effective time usable for task execution is computed by : $\text{WCAT}_B(T) = \frac{T - \text{OS}_{\text{overhead}} - T_{\text{min}}^{\text{preempt}}}{T}$ for the baseline cache and $\text{WCAT}_H(T) = \frac{T - \text{OS}_{\text{overhead}}}{T}$ for the HwCS, with T the period of the OS tick. Measurements are listed in Table I.

The measurements show that a system with the baseline cache have a worst case utilisation of 5% (although the average utilisation might be far better because we selected parameters to highlight the cost of preemptions with the baseline cache). Considering the worst case, almost all the available computing time is wasted in overheads, whereas with a system using the HwCS, the worst case available time is 90% in the same situation, which represent a huge improvement.

The Δ_S and Δ_R were also measured and show an decrease of a few percent related to the cache flush and reload time respectively, with a bigger transfer with the memory. The HwCS does not have to verify if data are cached when saving/restoring a context, thus saving some cycles for each cache line compared to the usual cache behaviour.

The only constraint is that $T_{\text{min}}^{\text{preempt}}$ must be lower than the tick period. The peak relative memory bandwidth is doubled with the HwCS as a layer can save/restore its content during another task execution.

Of course, further improvements are possible and will be implemented in future works. We could for instance unequally share the bus bandwidth between the layers or we could implement the save/restore Solution 2 described in Section III-B.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a new hardware component named HwCS, which makes the preemption overheads negligible in comparison to the task worst-case execution times. However, the HwCS imposes a delay Δ_R between the instant of the preemption decided in the theoretical schedule and the instant at which it actually happens on the platform. We proved that this behaviour of the HwCS can be taken into account in the real-time scheduling theory by simply reducing the deadline of each task by Δ_R time units. This property enables to reuse all the existing theory on the real-time scheduling while fairly comparing the scheduling algorithms.

Our experimental results show that the HwCS can potentially improve the worst case processor utilization from 5 to 90% for a 1ms OS tick on a uniprocessor platform using a two-layer HwCS and allowing preemptions every 1 ms. This performance improvement has a cost which is estimated to an increase of 3% for an Itanium processor die area.

The presented work is a first proof of concept and further improvements could be made in the design of the HwCS for uniprocessors systems. Furthermore, more comprehensive tests must still be conducted to measure the impact of the HwCS on the performances of various systems running realistic applications. Also, the impact of keeping the same memory area in a N-layer HwCS compared to the standard baseline cache is to be studied. The theory must be extended to multiprocessor platforms.

REFERENCES

[1] Kalray Corporation. (2013) A new generation of agile manycore processors & software solutions for high performance. [Online]. Available: http://www.kalray.eu

[2] ARM, *PL310 Cache Controller Technical Reference Manual*, 2007.

[3] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," *Real-Time Syst. Symp. 2008*, 2008.

[4] S. M. Petters and G. Färber, "Scheduling analysis with respect to hardware related preemption delay," in *Workshop on Real-Time Embedded Systems*, december 2001.

	BL 4kB	HwCS 2×4kB	BL 32kB	HwCS 2×32kB	HwCS 2×16kB
HwCS/cache alone:					
LUT6	1334	2900(×2.2)	1411	3186(×2.3)	3059(×2.2)
FF	1037	2105(×2.0)	1057	2224(×2.1)	2212(×2.1)
Logic (total)	2371	5005(×2.1)	2468	5410(×2.2)	5271(×2.1)
RAM (BRAM + 64b LUT)	1+53	2+106(×2)	12+36	18+384(×1.5/11)	10+200(×0.8/6)
Complete platform:					
LUT6	5294	6905(×1.3)	5372	7191(×1.3)	7063(×1.3)
FF	3208	4279(×1.3)	3228	4398(×1.4)	4386(×1.4)
Logic (total)	8502	11184(×1.3)	8600	11589(×1.35)	11449(×1.3)
RAM (BRAM + 64b LUT)	1+53	2+106(×2)	12+36	18+384(×1.5/11)	10+200(×0.8/6)
relative memory bandwidth (peak)	1	2	1	2	2
Cache flush (WC) [ms]	0.28	0.28	2.2	2.2	1.1
Cache reload (WC)[ms]	0.48	0.48	4.0	4.0	2.0
Δ_R [ms]	NA	0.45	NA	3.6	1.8
Δ_S [ms]	NA	0.26	NA	2.4	1.2
$T_{\text{preempt}}^{\min}$ [ms]	0.76	0.71	6.2	6	3
OS overhead (WC) [μs]	190	200	190	200	200
CPU WCAT@1ms [%]	5	90	NA	NA	NA
CPU WCAT@10ms [%]	90	98	36	98	98

TABLE I
HWCS/BASELINE CACHE PERFORMANCE AND COSTS FOR SEVERAL CONFIGURATIONS

[5] A. Bastoni, B. B. Brandenburg, and J. Anderson, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *OSPERT 2010*, S. M. Petters and P. Zijlstra, Eds., July 2010, pp. 33–44.

[6] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *RTSS 2010*, 2010, pp. 14–24.

[7] ——, "Is semi-partitioned scheduling practical?" in *ECRTS 2011*, July 2011, pp. 125–135.

[8] G. Nelissen, V. Nelis, J. Goossens, and D. Milojevic, "High-level simulation for enhanced context switching for real-time scheduling in mpsocs," in *JRWRTC 2009*, C. Seidner, Ed., 2009, pp. 47–50.

[9] M. Vetromille, L. Ost, C. Marcon, C. Reif, and F. Hessel, "RTOS Scheduler Implementation in Hardware and Software for Real Time Applications," *Seventeenth IEEE Int. Work. Rapid Syst. Prototyp.*, 2006.

[10] G. Chanteperdrix and R. Cochran, "The ARM fast context switch extension for Linux," *Real Time Linux Work.*, 2009.

[11] J. S. Snyder, D. B. Whalley, and T. P. Baker, "Fast context switches: Compiler and architectural support for preemptive scheduling," *Microprocessors and Microsystems*, vol. 19, pp. 35–42, 1995.

[12] K. Goossens, A. Azevedo, K. Chandrasekar, M. D. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha, "Virtual Execution Platforms for Mixed-Time-Criticality Applications: The CompSOC Architecture and Design Flow," *SIGBED Rev.*, vol. 10, no. 3, pp. 23–34, 2013.

[13] Microchip, *16-Bit Microcontrollers and Digital Signal Controllers with High-Speed PWM, USB and Advanced Analog*, 2012.

[14] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[15] SPARC International, *The SPARC architecture manual Version 9*, 1994.

[16] ARM, *ARM architecture reference manual*, 2007.

[17] Department of Computer Systems at the Tampere University of Technology, *COFFEE Core USER MANUAL*, 2007.

[18] Analog Devices, *SHARC Processor Programming Reference (Includes ADSP-2136x, ADSP-2137x, and ADSP-214xx Processors) Revision 2.2, March 2011*, 2011.

[19] N. Suzuki, S. Kurotaki, M. Suzuki, N. Kaneko, Y. Yamada, K. Deguchi, Y. Hasegawa, and H. Amano, "Implementing and evaluating stream applications on the dynamically reconfigurable processor," in *FCCM 2004*, 2004, pp. 328–329.

[20] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," *SIGARCH Comput. Arch. News*, vol. 18, no. 3b, pp. 1–6, 1990.

[21] Intel, *Intel® 64 and IA-32 Architectures. Software Developer's Manual. Volume 1-3C*, 2012.

[22] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, "Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 695–706, Aug. 2011.

[23] W. Grunewald and T. Ungerer, "Towards extremely fast context switching in a block-multithreaded processor," *EUROMICRO 96. Beyond 2000 . . .*, 1996.

[24] 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*, vol. 47, no. 6, pp. 700–713, june 1998.

[25] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *IPDPS '03*, 2003, pp. 114.1 (On CD–ROM).

[26] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS 2012*, 2012, pp. 13–23.

[27] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," *Real-Time Systems*, vol. 47, pp. 319–355, 2011.

[28] J. Reineke and D. Grund, "Sensitivity of Cache Replacement Policies," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 42:1—-42:18, 2013.

[29] S. Roy, "H-NMRU: An Efficient Cache Replacement Policy with Low Area," *International Journal of Parallel Programming*, vol. 38, no. 3-4, pp. 277–287, Feb. 2010.

[30] M. Chaudhuri, "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches," in *Proc. 42Nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 401–412.

[31] Z. Sustran, S. Stojanovic, O. Rakocevic, V. M. Milutinovic, M. Valero, and D. Computadores, "A Survey of Dual Data Cache Systems," pp. 450–456, 2012.

[32] J. Gee, M. Hill, D. Pnevmatikatos, and A. Smith, "Cache performance of the SPEC92 benchmark suite," *Micro, IEEE*, 1993.

[33] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. Sullivan, and T. Grutkowski, "The implementation of the Itanium 2 microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 11, pp. 1448–1460, 2002.

[34] D. Bradley, P. Mahoney, and B. Stackhouse, "6.6 The 16kB Single-Cycle Read Access Cache on a Next-Generation 64b Itanium Microprocessor," pp. 9–11, 2002.