PRETI: Partitioned REal-TIme shared cache for mixed-criticality real-time systems.

Benjamin Lesage, Isabelle Puaut, André Seznec

PRETI: Partitioned REal-TIme shared cache for mixed-criticality real-time systems.

Benjamin Lesage Isabelle Puaut André Seznec
{Benjamin.Lesage, Isabelle.Puaut}@irisa.fr / Andre.Seznec@inria.fr

Abstract

Multithreaded processors raise new opportunities as well as new issues in all application domains. In the context of real-time applications, it has created one major opportunity and one major difficulty. On the one hand, the concurrent execution of multiple threads creates the opportunity to mix, on the same hardware platform, the execution of a complex real-time workload and the execution of non-critical applications. On the other hand, for real-time tasks, timing deadlines must be met and enforced. Hardware resource sharing, inherent to multithreading, hinders the timing analysis of concurrent tasks. Two different objectives are pursued in this work: enforcing timing deadlines for real-time tasks, and achieving the highest possible performance for the non-critical workload.

In this paper, we present the PRETI, Partitioned REal-TIme shared cache scheme, a flexible hardware-based cache partitioning scheme that aims at pursuing these two objectives at the same time. Plainly considering inter-task conflicts on shared caches for real-time tasks yields very pessimistic timing estimates. We remove this pessimism by allocating private cache space for real-time tasks. During the execution of a real-time task, our scheme reserves a fixed number of cache lines per set for the task. Therefore, uniprocessor, i.e. unithread, worst-case execution time (WCET) estimation techniques can be used, resulting in tight WCET estimates. Apart from the private spaces reserved for the real-time tasks currently running, the remaining cache space is shared by all tasks running on the processor, in particular non-critical tasks, enabling high performance for these tasks. While the PRETI cache scheme is of great help to achieving our both objectives, its hardware implementation consists only in a slight modification of the LRU cache replacement policy.

Experiments are presented to show that the PRETI cache scheme allows for both guaranteeing the schedulability of a set of real-time tasks with tight timing constraints, and enabling high performance for the non-critical tasks.

1 Introduction

Need for performance, as well as the economic pressure to use off-the-shelf components or standard IPs, lead to the use of standard microarchitectures in a large range of application domains. Most processors are designed to achieve the best average performance as possible on as large as possible application domains.

The use of multithreaded processors in the real-time systems domain creates the opportunity to mix on the same platform the execution of critical real-time tasks with a non-critical workload. However, using multithreaded processors to execute critical real-time tasks creates at the same time a major difficulty, since it must be guaranteed that all critical tasks meet their timing constraints in all cases, including the worst-case. This *safety* property of the *worst-case execution time* (WCET) estimate is of particular importance for hard real-time systems, e.g. control of airplanes: one has to guarantee that the effective execution time will always be smaller than the WCET estimation. Although research on WCET estimation on uniprocessor featuring instruction level parallelism and complex memory hierarchy has made important progress in the last two decades [20], research in the domain is still going to face the ever increasing complexity of hardware. In general, the more complex is the processor microarchitecture, the less precise are estimated WCETs. In the context of multithreaded processors, the major source of pessimism in WCET estimation comes from the sharing of the cache between hardware threads.

While meeting deadlines is the objective for critical real-time tasks, achieving high performance is the goal for non-critical workloads. Ideally, one would like to address these two, apparently opposite, objectives with a standard microprocessor. Moreover the proposal of a new microarchitectural feature will not be considered by the industry if it only targets real-time systems unless it induces very marginal modification of the microarchitecture. In this paper, we propose a very simple modification of the replacement policy in the shared cache of a multithreaded processor that both addresses meeting strict deadlines in real-time systems and achieving high performance on non-critical workloads.

On a multithreaded processor, the threads share the hardware core resources and the memory hierarchy, in particular the first level cache. Functional units sharing can create difficulties to compute a very tight

(within a few cycles) WCET estimates for each basic block on superscalar processors [11]. However the actual estimate is much more sensitive to the worst-case cache behavior, where a hit or a miss may result in hundreds of cycles difference on the execution time of a short basic block. Therefore, in this paper we will assume a very simple and predictable model for the execution core, whereas we will concentrate our efforts on (i) guaranteeing deadlines through worst-case analysis of the cache behavior for critical tasks and (ii) improving cache behavior for non-critical tasks.

In this paper, we present the PRETI, Partitioned REal-TIme shared cache scheme, a flexible hardware-based cache partitioning scheme that mitigates the rigidity of classic cache partitioning schemes. In order to avoid very pessimistic timing estimates associated with inter-task conflicts on real-time critical tasks, we create a private cache space for these tasks. All along the execution of a real-time task, our scheme reserves a fixed number of cache lines per set for the task. When a task is allocated N lines per set, the PRETI scheme does not allocate a fixed subset of N ways for the task, but instead guarantees that for each set, the N most recently used blocks from the task, mapped in this set, are present in the cache, regardless of their actual assigned ways in the cache. This guarantees that the analysis of the worst-case cache behavior of the task, assuming for a N-way set-associative cache, is safe. Apart from the private spaces reserved for the real-time tasks currently running, the cache space is shared by all tasks running on the processor, i.e. non-critical tasks, but also the real-time tasks for their least recently used memory blocks. This allows for enhancing the performance of the non-critical threads and also for reducing the effective execution time of the real-time tasks. In practice, the PRETI cache scheme is implemented as a slight modification of a conventional LRU replacement policy.

In summary the PRETI cache scheme:

- allows providing *predictability guarantees* for critical real-time tasks, through reserved space in the shared cache for these tasks
- allows as good as possible *performance for non-critical tasks*, through the dynamic allocation of cache lines not currently used by critical tasks,
- is a *marginal modification of conventional LRU replacement*, and as such could be considered by the industry as an architectural support for real-time applications

In this paper, hard real-time systems, in which safety is critical, are considered to illustrate the PRETI

cache. The PRETI cache scheme exhibits the same benefits in the context of soft real-time systems where the safety of WCET estimates can be relaxed.

The remainder of this paper is organized as follows. Section 2 surveys the related works on using caches in real-time systems. The PRETI cache scheme is detailed Section 3 including considering hardware implementation and WCET analysis aspects. Section 4 gives the results of experimentations conducted to illustrate the behavior of our proposal. Finally, Section 5 summarizes the present study and gives directions for future works.

2 Related work

Sharing caches on real-time systems has essentially been considered for multi-cores or multiprocessors, or in the context of time shared-multiprocessors. To the best of our knowledge, no specific studies has addressed sharing caches in multithreaded processors on real time caches. However, most of the research done in the context of multiprocessors or multicores is directly relevant.

For multi-core architectures, state-of-the-art shared cache analyzes [12, 5], stemming from abstract interpretation based analyzes [3], have raised the problems of inter-task conflicts and their negative impact on tasks' predictability. The use of bypass [5], information about tasks lifetimes [12], or the conjoint use of partitioning and locking [17] have been recommended to reduce the impact of these conflicts. Similarly to these works, our proposal aims at containing inter-task conflicts, hence improving systems predictability.

Cache partitioning, in the context of real-time systems, has been mostly explored for preemptive systems, to eliminate cache related preemption delays [18, 21, 13]. The advent of multi-processors and multi-core architectures has led to the use of partitioning for shared caches, to preclude inter-task conflicts in such concurrent systems [9, 17].

The partitioning schemes introduced, or assumed, in related work [21, 13, 9, 17] tend to be shackled by their inherent strictness. Tasks' memory blocks are kept within the limits of their allocated partitions, and conversely, a partition in the cache can only be modified by its owner. Our proposal alleviates some of these restrictions, to maximize cache space utilization, and allows for reasonable average-case performances.

Prioritized caches [18] were proposed as a mean to advantage the highest priority tasks in the context of preemptive systems. A cache line can only be redeemed by a task of priority higher than the current owner of the line. Hence, the higher the task priority, the less it suffers from inter-task conflicts. Furthermore, resources are allocated to tasks on demand. This proposal allows to mix critical and no-critical tasks. We

use a similar on-demand cache line allocation mechanism. Unlike prioritized caches, using the proposed PRETI scheme, partitions' sizes are upper-bounded to prevent the trashing of the whole cache by high priority tasks.

A software-based per-set partitioning has also been proposed for preemptive real-time systems [21, 13]. Using the compiler, the memory mapping of the tasks of the system is altered to ensure their spatial isolation in the cache. However, altering tasks' mapping in the memory is far from trivial; significant modifications of the compilation tool chain are required to dispatch tasks' memory blocks to appropriate addresses in the memory, depending on the cache sets where they belong. Instead, we propose a simple hardware-based partitioning scheme.

S.M.A.R.T. caching [9] addresses the additional issue of shared data structures. Using this hybrid hardware and software scheme, cache sets are grouped into partitions and a special, statically-sized partition, the *shared pool*, is used to cache data shared between multiple tasks, and bear less critical tasks. The partitioning scheme proposed in this paper uses a similar *shared space* to accommodate for additional tasks. However, our *shared space* grows and shrinks dynamically as resources are freed or allocated.

The per-set division of the cache, shared by compile-time methods [21, 13] and S.M.A.R.T. caching [9], allows for fine-grained partitions. However, disused resources are beyond reclaim; merging set-based partitions, without flushing a part of the cache contents, could induce the duplication of memory blocks in different sets (their previous set and the post-merge one). With the PRETI scheme, we rely on a "per-way" division of the cache instead. However, this "per-way" division is virtual in the sense that the PRETI scheme guarantees that for a given task and for each set, at least the N most recently used blocks reside in the cache, but it does not impose that these blocks are stored in fixed ways.

The PRETI cache scheme has been defined for real-time systems mixing real-time tasks and non–critical tasks. It benefits from the predictability increase that can be expected from the use of partitioning for shared caches in the context of multi-core architectures [9], while preserving flexibility, in the sense that unclaimed cache space can be used by any task, be it real-time or not.

3 The PRETI cache partitioning scheme

The PRETI (Partitioned REal-TIme) shared cache is a *per-way* partitioning scheme intended for shared cache levels of the memory hierarchy. Unlike existing strict partitioning schemes, by *per-way*, we mean that, for a given task featuring a N-way allocated partition, and for each cache set, at least the N most

recently used blocks are present in the cache. In the context of multithreaded architectures and real time systems, the first objective of the PRETI cache is to ensure the real-time tasks' timing predictability, through guarantees in terms of cache space available for these the critical real-time tasks. Secondly, PRETI also aims at maximizing the overall performance of the system, particularly non-critical tasks.

In the following, the behavior of the PRETI cache is first introduced (§ 3.1). Based on this description of the mechanism, possible implementations (§ 3.2), then static cache analyzes targeting the PRETI cache are discussed (§ 3.3).

3.1 Principles

The principles introduced subsequently regulate the behavior of PRETI caches, comprised of the notions which underlie the partitioning of the cache lines and the cache replacement policy.

Private and shared partitions divide the cache in distinct spaces. At creation, a task (typically a real-time task) can be allocated a *private space* consisting in a fixed number of ways in the set-associative cache. Each *private space* is used by a single task, its owner, to store its most recently accessed memory blocks. This private space is *virtual* in the sense that to avoid moving blocks in the cache, the private space of N ways consists in the N most recently used blocks accessed by the task in each cache set regardless of their physical location; their effective storage place may vary during the task execution. Upon task termination, the associated private space is released.

The *shared* space, on the other hand, keeps blocks for *all* the tasks accessing the cache, whether they have a dedicated private space or not. It holds tasks' least recently accessed memory blocks. The shared space is dynamically composed from all the cache lines that do not belong to any private space: (i) cache lines occupied by blocks from tasks that do not own any private cache space, and (ii) cache lines used by a task beyond its private space allocated capacity.

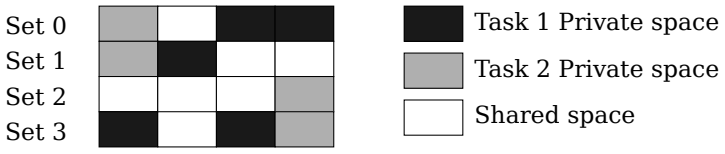


Figure 1: Division of a 4-way cache between shared and private spaces during execution.

Figure 1 illustrates the division of a 4-way cache between shared and private spaces during the execution of two tasks, Task 1 and Task 2. Task 1 has been allocated a maximum of two cache ways, whereas Task 2 can claim the ownership of at most one way. Black (resp. grey) blocks depict blocks from the private space

of Task 1 (resp. Task 2), whereas white blocks are blocks from the shared space. In Set 0, Task 1 and Task 2 fully use their private partition; the shared space of Set 0 is then reduced to one cache line, the statically unallocated one. In Set 1, Task 1 has only claimed a single line in its private space, whereas Task 2 fully uses its private space; the Set 1 shared space is composed of two cache lines, the unallocated one and one line unclaimed by Task 1 at this time instant.

This example also illustrates an interesting property of the PRETI cache: real-time task are often small tasks, that sometimes do not touch every cache line before their termination. When a real-time task does not claim its overall private space, the scheme dynamically enables the other tasks (non-critical tasks or real-time tasks) to use this cache space.

The cache replacement policy is directly derived from the Least Recently Used (LRU) policy. The LRU order is maintained among all blocks present in a cache set. However, in case of a miss, the LRU block is not systematically evicted, to guarantee a minimum reserved capacity in the cache to tasks with private partitions. In case of a miss, the block to be replaced is the oldest block that:

- either belongs to the shared space,
- or belongs to the private space of the task triggering the miss

The block is systematically inserted at the Most Recently Used position in the set.

This replacement policy, for a task with a N-way private space, enforces the property that each set in the cache maintains at least the N most recently blocks used by the task. From the task perspective, it means that the private space acts as a N-way LRU set-associative cache. This property is of particular interest for hard real-time systems because of the timing predictability properties of LRU replacement, as demonstrated in [16].

3.2 Implementation of a PRETI cache

The hardware overhead of PRETI caches over a conventional set-associative LRU cache is only on the replacement policy. The access hit time is not modified.

On a hit, the LRU tags must be modified as on a conventional LRU cache. On a miss, the selection of the replaced block is different from the one on the LRU cache. In order to implement this selection, an extra information must be associated with the cache block, the *task id* of the owner task, updated on an access to the block. This is illustrated on Figure 2. The tasks having a private partition (typically real-time tasks) have

distinct *task ids*, whereas the other tasks (typically non-critical tasks) may share the same task identifier.

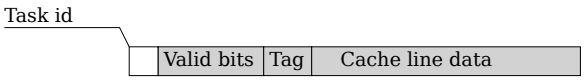


Figure 2: Storage requirements for PRETI caches: each cache line is extended with a *task id*.

The cost, in terms of storage requirements, of the proposed implementation is low: $\lceil log_2(T+1) \rceil * \ell$ bits for the *task ids*, where ℓ is the number of cache lines and T the maximum number of distinct private spaces which may reside in the cache at the same time. On the other hand, some extra logic is needed to perform the selection of the replaced block, but this logic is limited and is not on the critical path.

3.3 Analyzing PRETI cache behavior on a real-time task

As the PRETI cache is designed to provide guarantees for real-time tasks, static analysis will be used to compute WCET estimates for such tasks. WCET estimation requires to identify tasks' worst-case behavior with regard to the memory hierarchy.

In the context of a PRETI cache, the partitioning scheme enforces strong properties that enable static worst-case analysis of real-time tasks. First, by construction, private spaces, used to store tasks' most recently accessed memory blocks, can only be altered by their respective owner; they are similar to small private caches in the sense they are free of inter-task conflicts. Furthermore, each task's private space is statically known to be the very least amount of cache space available to this task during execution, and replacement policy-wise, private spaces mimic LRU-enforcing private caches.

The analysis of a task's behavior can thus be safely reduced to the analysis of said task's private space. Such a private space analysis can be performed using state of the art private cache contents analyzes [19, 6, 10], while assuming a LRU cache of the same dimensions than the task's statically bounded private space. With respect to WCET analysis, PRETI and strictly-partitioned caches behave the same way.

Nevertheless, it should be noted that sharing data among tasks causes difficulties in the worst-case analysis of the cache content. Experiments conducted by previous studies on WCET estimation for multi-core architectures with shared caches [12, 5] show that data sharing yield to pessimistic WCET estimates; this is because data sharing may result in both destructive and constructive effects on the shared cache, both of them being hard to predict statically. The same difficulty applies when using the PRETI cache. A possible solution in the context of PRETI cache would be to avoid tagging shared data with the private tag of the task.

4 Experimental results

In this section, after a presentation of the experimental setup (§ 4.1), we illustrate the ability of the PRETI partitioning scheme, for mixed-criticality workloads, to both guarantee the schedulability of real-time task sets (§ 4.2), and to enable high performances for the non-critical tasks (§ 4.3).

4.1 Experimental setup

Simulated hardware platform For the purpose of our experiments, we will consider a very simple model of time multithreaded processor. 3 hardware threads are assumed and each thread is granted the access to the hardware every 3 cycles in a round-robin fashion.

The processor executes an instruction per cycle and the execution of a thread is stalled on any cache miss until the miss is resolved. The processor features a 8-way set-associative 4KB data cache and a same-sized 8-way set-associative instruction cache. 32 bytes cache blocks are used and LRU replacement policy is assumed for both caches. Upon a miss, the miss penalty of 150 cycles is observed by the thread, but the memory can service up to 6 misses concurrently, one data miss and one instruction miss per thread. The caches sizes correspond to the use of a small embedded processor, suited to the assumed workloads.

Software environment Each task, be it time-critical or not, is statically allocated to a single hardware thread.

Then, on each hardware thread, a non-preemptive Earliest Deadline First scheduling (NP-EDF [8]) is used for time-critical tasks. Upon each release of a time-critical task, tasks are sorted by increasing deadlines. If no real-time task is currently running on the hardware thread, the released task with the earliest deadline is executed first, without preemption, until completion. The non-preemptive scheduling for critical tasks eliminate cache-related preemption delays from impacting the performances of time-critical tasks and allows for static WCET estimation.

Non-critical tasks execute in the background; they are allowed to execute on their assigned hardware thread only to when no time-critical task is ready, and are preempted as soon as a time-critical task is released.

Cache analysis and WCET estimation The presented experiments have been conducted on MIPS R2000/R3000 binary code compiled with gcc 4.1 without any code optimization and using the default linker memory lay-

out. WCET estimates of real-time tasks were computed using a state-of-the-art static WCET estimation tool [**?**] [1] using the standard *Implicit Path Enumeration Technique* (IPET) technique for WCET computation [20]. Memory references were classified using the most recent static cache analysis techniques presented in [6, 2]; every reference is classified as *hit* if the block is for sure in the cache, and as a *miss* otherwise. Focusing on caches WCET estimates only take their contribution into account.

Timing simulations Simulations have been performed using trace driven simulations. A simple timing model is used: an instruction is fetched and executed per cycle, but the execution is stalled upon any cache miss. For the sake of simplicity, we assume that the execution overhead of the task scheduler is negligible. Moreover, a unique memory access trace per benchmark is used.

Benchmarks Unfortunately, very few realistic real-time benchmarks are available for the research community; only benchmarks with limited execution time and data and instruction footprints are publicly available. Consequently, our experiments were conducted using the benchmark sets that are used in the context of research on hard real-time systems, in particular for the context of WCET estimation [4].

The experiments were conducted using two small but standalone real-time applications, named Debie and Papabench, as well as benchmarks from the WCET benchmark suite maintained by the Mälardalen WCET research group. The Debie software [7], developed by Space Systems Finland, monitors the impact of space debris and micro-meteoroids using electrical and mechanical sensors. The Papabench tasks [14] come from the Paparazzi project [1], used to build autonomous unmanned aerial vehicles.

In our experiments, the tasks from the Debie and Papabench real-time benchmarks are used as the real-time tasks of the studied workloads. The real-time tasks are released periodically, with a deadline equal to the task period. The non-critical tasks were selected among the ones maintained by the Mälardalen WCET research group. Non-critical tasks are not periodic and use all slack time left by real-time tasks, if any.

The characteristics of each task are exposed in Table 1 (from left to right are the sizes in bytes of *code*, *data*, *bss* -uninitialized data and maximum *stack* sections, and execution *frequency* for the periodic, real-time tasks).

Task sets Three different sets of periodic time-critical tasks have been composed from the tasks presented in Table 1. The resulting task sets are given in Table 2. The *debie* task set is composed of all tasks of

[1] Reference omitted for blind review

Task	Code (bytes)	Data (bytes)	Bss (bytes)	Stack (bytes)	Frequency
Debie					
acquisition_task	5968	1280	101904	200	100Hz
hit_trigger_handler	2408	647	218	104	400Hz
monitoring_task	12372	667	65772	296	1Hz
tc_execution_task	12928	1280	101904	200	100Hz
tc_interrupt_handler	2800	134	35648	64	1000Hz
tm_interrupt_handler	872	23	168	56	1000Hz
Papabench					
task 1	3960	74	78	32	40Hz
task 2	696	52	242	32	40Hz
task 3	1492	64	66	32	20Hz
task 4	372	4532	568	32	20Hz
task 5	1440	4257	20	32	20Hz
task 6	8508	572	302	56	40Hz
task 7	1352	194	158	32	20Hz
task 8	264	212	46	8	20Hz
task 9	10284	552	511	88	4Hz
task 10	33700	8192	302	1160	4Hz
task 11	268	90	0	32	4Hz
task 12	1120	145	0	32	4Hz
task 13	21176	284	255	0	10Hz
Mälardalen					
adpcm	7872	1300	436	152	—
fft	3772	88	128	296	—
lms	2992	248	1092	184	—

Table 1: Benchmark characteristics

the Debie benchmark. The *papabench.manual* and *papabench.auto* task sets correspond respectively to a remote-controlled and an automatic operating mode of a drone.

Task set	Tasks
debie	acquisition_task, hit_trigger_handler, monitoring_task, tc_execution_task, tc_interrupt_handler, tm_interrupt_handler
papabench.manual	tasks 1, 2, 3, 4, 5, 6, 7, 8, and 13
papabench.auto	tasks 3, 4, 5, 7, 8, 9, 10, 11, 12, and 13

Table 2: Task sets

4.2 Impact of cache partitioning on schedulability

In this section, we illustrate the benefits of using cache partitioning to allow for the schedulability of real-time tasks with tight timing constraints. For each real-time workload, we determine the minimum

functioning frequency of the system for which the real-time workload would be safely schedulable. By safely schedulable, we mean that there exists a cache partitioning, a task allocation and a schedule such that even in the unrealistic case where the execution time of every task is equal to the WCET estimate computed by the static analysis, all the timing constraints are met.

To compute this minimum frequency, we determine the best pair (cache partitioning, task allocation) among the threads through a systematic exploration of the possible (cache partitioning, task allocation) pairs. Assuming a non-preemptive system (i.e., no real-time task can be interrupted), when the WCETs and the timing constraints of a set of tasks are known, it has been showed in [8] that the system can be scheduled on a uniprocessor if 1) the WCET of any task is shorter than the delay between two consecutive deadlines and if 2) when a task is launched, it can complete before the deadline of any other task assigned to the same thread. This is a sufficient and necessary condition.

We determine this best pair (cache partitioning, task allocation) for the PRETI cache as well as the best task allocation for the shared cache. These best configurations are illustrated on Table 3. We also illustrate for each hardware thread the predicted number of instructions that are executed per second on its assigned real-time subset task subset. As mentioned in the introduction, the contribution of the memory hierarchy to the WCET is very large (tens of cycle per instruction) and plainly justify our simple processor model which essentially ignores the contributions of instructions dependencies and pipeline hazards. It is noticeable that assuming a fully shared cache leads to a minimum frequency for guaranteeing schedulability much higher respectively (13x, 5.8x and 5.6x) than the minimum frequency needed for the PRETI cache.

Note that the best pair (cache partitioning, task allocation) does not always allocate the 8 cache ways among the three hardware threads. This is due to the WCET analysis that sometimes does not lead to better WCET estimates when allocating more ways to a task. In the context of PRETI cache, allocating a smaller number of ways in the private cache space allows increasing the shared space and therefore may enable higher performance on the non-critical tasks.

4.3 Performances of non-critical tasks using PRETI

In this section, we illustrate the ability of the PRETI partitioning scheme, for mixed-criticality workloads, to enable high performances for the non-critical tasks while guaranteeing schedulability.

In order to illustrate this property we run simulations assuming 3 different cache configurations, the PRETI cache, a shared cache and a completely partitioned cache. We run 3 different sets of simulations

debie using partitioned caches
(1.5Ghz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	monitoring_task	acquisition_task hit_trigger_handler tc_execution_task tc_interrupt_handler tm_interrupt_handler	— —
Instruction cache partition	7 ways	1 way	—
Data cache partition	4 ways	3 ways	—
Real-time tasks' instructions per second	54,109,612	18,365,000	—

debie using shared caches
(19.5Ghz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	monitoring_task	acquisition_task hit_trigger_handler tc_execution_task tn_interrupt_handler	tc_interrupt_handler
Real-time tasks' instructions per second	60,031,033	18,146,700	372,000

papabench.auto using partitioned caches
(3.0Mhz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	3, 4, 5, 7, 8, 11, 12 & 13	9 & 10	—
Instruction cache partition	1 way	1 way	—
Data cache partition	3 ways	1 way	—
Real-time tasks' instructions per second	90,352	53,672	—

papabench.auto using shared caches
(17.5Mhz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	3, 4, 5, 7, 8, 12 & 13	9, 10 & 11	—
Real-time tasks' instructions per second	91,024	42,164	—

papabench.manual using partitioned caches
(3.1Mhz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	3, 4, 5, 7, 8, & 13	1 & 2	6
Instruction cache partition	1 way	1 way	1 way
Data cache partition	1 way	0 way	2 ways
Real-time tasks' instructions per second	89,760	49,920	74,000

papabench.manual using shared caches
(17.5Mhz processor frequency)

	Thread 0	Thread 1	Thread 2
Allocated tasks	3, 4, 5, 7, 8, & 13	1 & 2	6
Real-time tasks' instructions per second	90,260	49,920	74,560

Table 3: Best configurations for the *debie*, *papabench.auto* and *papabench.manual* under the best partitioning or using shared caches.

on which we fix the frequency of the processor, respectively to 1, 2 and 3 times the minimum frequency necessary to schedule the real-time workload on the PRETI cache. The system is run for 1.5 billion cycles for papabench.auto and papabench.manual and for 1 second of execution i.e one complete execution period for debie (i.e. respectively 1.5 billion, 3 billion and 4.5 billions cycles).

1. For the PRETI cache, we choose the best pair (cache partitioning, task allocation) obtained above for each workload, i.e. the pair that requires the minimum frequency to get the workload schedulable.

2. The shared cache should not be used in a hard real-time system , since there is no safety guarantee for the frequency that we will simulate. This shared cache where is presented in order to illustrate the performance that could be reached on non-critical tasks if there were no safety constraint on the real-time tasks.

3. By completely partitioned cache, we denote a cache where each hardware thread will have a fixed set partition during the complete execution, that is a hardware thread cannot access the blocks from the other hardware threads. With a completely partitioned cache, assuming the same task allocation as for the PRETI cache, the safety constraints on the real-time tasks will be respected if the partition sizes are larger or equal that the ones for the PRETI cache [2]. In our experiments, we allocate all the cache ways among the hardware threads(see Table 4).

		Thread 0	Thread 1	Thread 2
debie	Instruction cache partition	7 ways	1 way	0 way
	Data cache partition	4 ways	3 ways	1 way
papabench.auto	Instruction cache partition	3 ways	3 ways	2 ways
	Data cache partition	3 ways	3 ways	2 ways
papabench.manual	Instruction cache partition	3 ways	2 ways	3 ways
	Data cache partition	3 ways	2 ways	3 ways

Table 4: Cache partitions allocated to the hardware threads under completely partitioned caches, for the debie, papabench.auto and papabench.manual task sets.

As a first round of simulations, on each hardware thread, when not running one critical task, we execute repetitively a non-critical task. The respective non-critical tasks executed are lms, adpcm and fft on the three hardware contexts. We measured the performance of each non-critical task in Instruction Per Cycle (IPC)

[2] As mentioned in the previous section, on the PRETI cache, we do not need to allocate all the cache ways to private spaces, as long as the timing constraints are respected

over the whole execution time. The maximum achievable performance for the thread would be 1 instruction every 3 cycles if there were no cache miss and no interruption by the critical tasks.

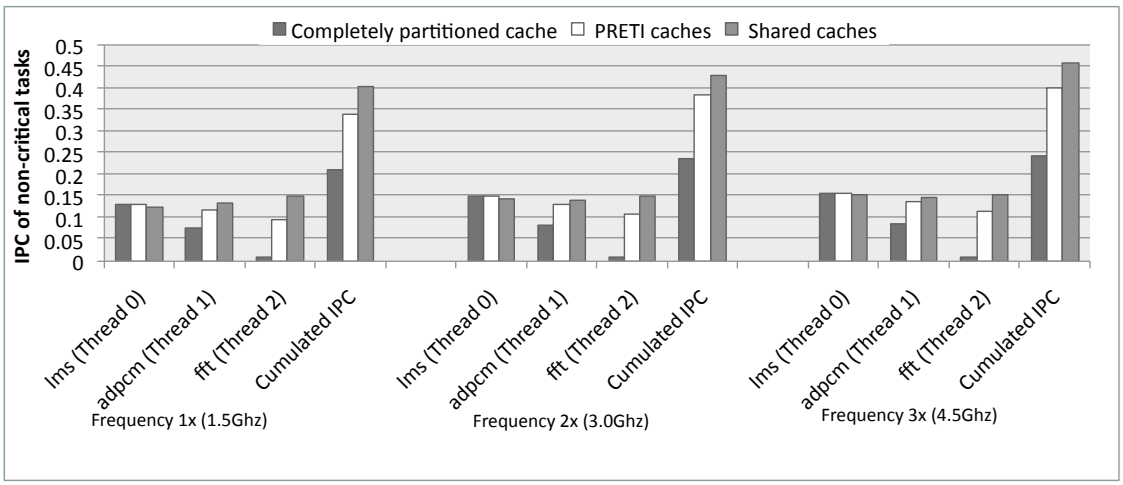
The simulation results are provided in Figure 3 and illustrate the benefits of sharing caches for the non-critical tasks. The results are provided for the debie, papabench.auto and papabench.manual, from top to bottom. For each workload, from left to right, the frequency of the processor was fixed to respectively 1, 2 and 3 times the minimum frequency necessary to schedule the real-time workload on the PRETI cache. The cumulated IPC is the sum of the IPC of the non-critical tasks running on each hardware thread.

Clearly always partitioning the cache is a major handicap for Thread 2 and on a less extent for Thread 1on the debie workload: the ways of the caches are mainly reserved to guarantee the safety on Thread 0. On debie, the PRETI cache is able to share the cache space for a large portion of the time (when the Thread 1 is not running the real-time workload). On Thread 1, for the three simulations, the execution time of the real-time tasks is in the range of 350 million cycles for the PRETI cache per second. That is the overall cache is completely shared for respectively for 77 %, 88 % and 92 % of the time for the three simulated frequencies. Therefore the performance on the non-critical tasks is in the same range when using the PRETI cache and the shared cache. The same phenomenon also occurs on the two papabench workloads. On papabench.manual, the performance on the non-critical workloads are very close for the PRETI cache and the shared cache for all threads while performance on Thread1 and Thread 2 is much lower for the partitioned cache.

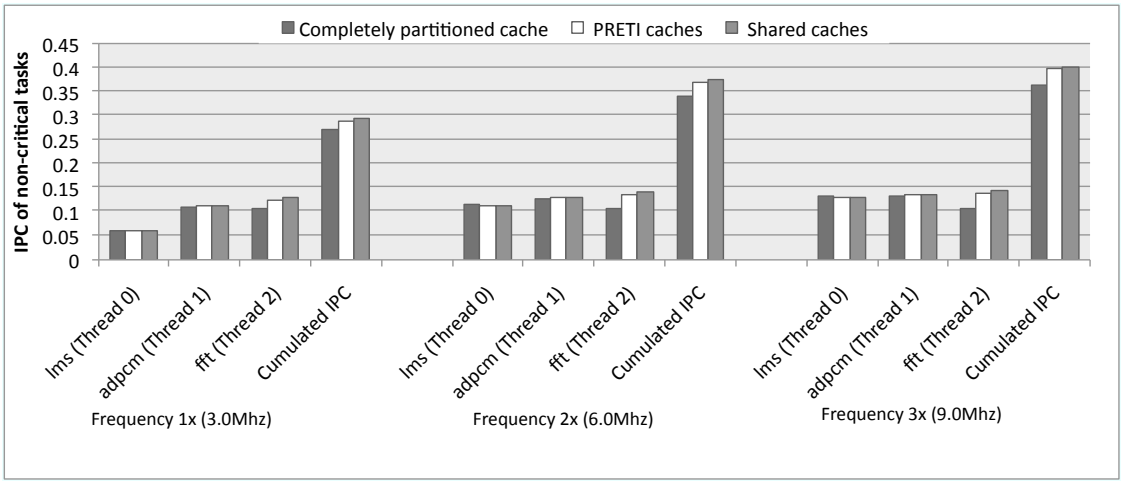
Notice that the lms task, running on Thread 0, has a particular behavior. It performs slightly worse when using shared or PRETI caches, than when using the partitioned cache. Its blocks tend to be evicted from the shared space by the other tasks because it exhibit lower temporal locality. On the other hand, the performance of the other tasks improve when sharing the cache space (shared and PRETI cache).

As a second round of simulations, we only execute one non-critical task concurrently with the real-time workload. This non-critical task is scheduled on the first hardware thread that becomes available. We run 3 sets of simulations using respectively lms, adpcm and fft as this single non-critical task. Simulation results in Figure 4 globally illustrates that the PRETI cache structure allows the non-critical task to benefit from shared cache space. The IPC of the non-critical task is presented by processor frequency, from top to bottom, then by concurrent real-time workload and non-critical task.

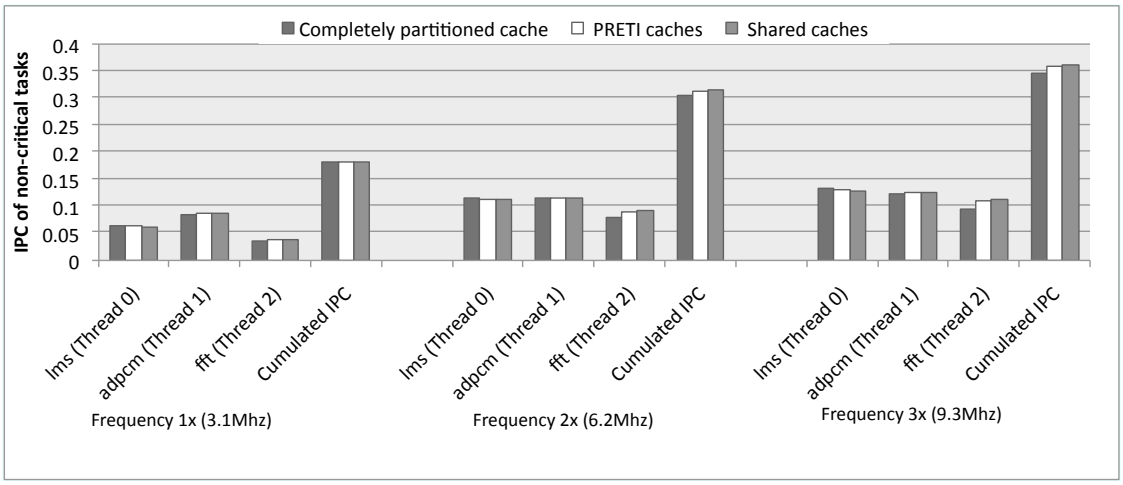
Completely partitioning the cache results in a worse cache behavior than using a fully shared cache, especially at higher frequencies, 2x and 3x. As stated earlier, using PRETI caches, the overall cache is completely shared for more than 77% of the time for each simulation. Therefore, as frequencies increases,



(a) Performances of the non-critical tasks alongside the debie workload.



(b) Performances of the non-critical tasks alongside the papabench.auto workload.



(c) Performances of the non-critical tasks alongside the papabench.manual workload.

Figure 3: IPC of the non-critical tasks, allocated to different threads, with the real-time workloads.

the gap between the performance of the non-critical task using completely partitioned caches, and their performance using shared caches, increase. Conversely, the gap between the performance of PRETI caches and shared caches shrinks.

As when running 3 threads, the sensitivity of the lms benchmark to sharing the cache space with other tasks induces some particular behavior: on papabench.auto and for frequency 1x, Thread 0 is busy for 64 % of the time with the real-time tasks, this induces a slight performance degradation for lms for shared cache and PRETI cache compared with the partitioned cache. For higher frequencies, lms runs alone for a longer time and the use of shared or PRETI cache on this interval compensates for the performance loss encountered when sharing.

5 Conclusion

During the past decade, the need for performance and the tremendous integration technology progress have allowed inexpensive superscalar processors featuring instruction level parallelism as well as memory hierarchy. Moreover, nowadays, these processors targeting large scale volumes are also featuring hardware thread level parallelism. Economic pressure is pushing for the use of these standard processor components or IPs in every application domain. As a consequence, an application domain can only marginally influence the design of the standard processors. That is the processor manufacturers will consider adding a feature to a standard processor only if the application domain generates huge volume, or if the modification is very marginal and does not impair the rest of the design. This applies for real-time systems.

Standard processors are optimized for average case performance. Unfortunately, real-time systems must be designed to enforce timing constraints in all situations, including the worst-case scenarios. The increase of performance of standard processors has been coupled with an explosion of the complexity (instruction level parallelism, memory hierarchy and thread level parallelism). This complexity has been widening the gap between the effective average run-time execution time of real-time tasks and the WCET estimate that can be safely obtained through static analysis. Every increase in hardware complexity (memory hierarchy, shared cache levels) has resulted in a larger gap between actual performance and predicted worst-case performance [6]. This has lead some authors to propose to completely avoid cache sharing, and instead rely on only private caches or equivalently strictly partitioning schemes [15]. On the other hand, the use of standard processors in real time systems has raised the possibility to execute non-critical tasks concurrently with critical tasks on the same platform. These non-critical tasks are not concerned with meeting deadlines

but with maximizing their performance: for these non-critical tasks cache sharing is helping to ensure high performance.

In this paper, we have proposed a small adaptation of the cache replacement policy on a shared cache for a multithreaded processor, the PRETI cache scheme. The PRETI cache scheme is a small variation of a shared LRU cache scheme. But this small modification can be of major benefit for using the multithreaded processor in a real-time system. By allowing critical real-time tasks to grab a private cache space, the PRETI cache allows WCET analysis to produce WCET estimates that are tighter than if a shared cache was used. Through this slight modification, we become able to guarantee the schedulability of some real-time workloads that would not be schedulable using a shared cache. By releasing this private space to all the tasks upon the critical task termination, the PRETI cache enables high performance on all the tasks, i.e. performance closer to the one obtained on a completely shared LRU cache.

While our study and experiments have been done in the context of hard real-time systems, where missing a constraint can be safety critical (e.g. controlling a car airbag), the PRETI cache scheme could also be used in soft real-time systems where missing a constraint is less critical (e.g. missing a frame in a video). As for hard real-time systems, the PRETI cache scheme can provide the soft real-time task with a private cache space that isolates its working set from interferences from other tasks, thus providing the same cache behavior predictability as a private cache. It also allows the tasks to benefit from the overall cache space when the real-time task is finished.

References

[1] www.recherche.enac.fr/paparazzi.

[2] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for WCET analysis and layout optimizations. In *IEEE Real-time Systems Symposium (RTSS).*, 2009.

[3] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, 1999.

[4] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen WCET benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010.

[5] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th Real-Time Systems Symposium*, pages 68–77, Washington D.C., USA, Dec. 2009.

[6] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 456–466. IEEE Computer Society, 2008.

[7] N. Holsti, T. Lngbacka, and S. Saarinen. Using a worst-case execution time tool for real-time verification of the debie software. In *Proceedings of the DASIA 2000 (Data Systems in Aerospace) Conference*, pages 92–9092, 2000.

[8] K. Jeffay, D. Stanat, and C. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129 –139, dec 1991.

[9] D. Kirk. Smart (strategic memory allocation for real-time) cache design. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 229–237, Dec 1989.

[10] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In N. Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[11] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Syst.*, 34(3):195–227, 2006.

[12] Y. Li, V. Suhendra, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-time System Symposium (RTSS)*, 2009.

[13] F. Mueller. Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, LCTES '95, pages 125–133, New York, NY, USA, 1995. ACM.

[14] F. Nemer, H. Cass, P. Sainrat, J.-P. Bahsoun, and M. De Michiel. PapaBench : A Free Real-Time Benchmark. In F. Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET), Dresden, 04/07/2006*, page (on line), http://drops.dagstuhl.de/, juillet 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[15] M. Paolieri, E. Qui andones, F. Cazorla, R. Davis, and M. Valero. Ia3: An interference aware allocation algorithm for multicore hard real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 280 –290, april 2011.

[16] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37:99–122, November 2007.

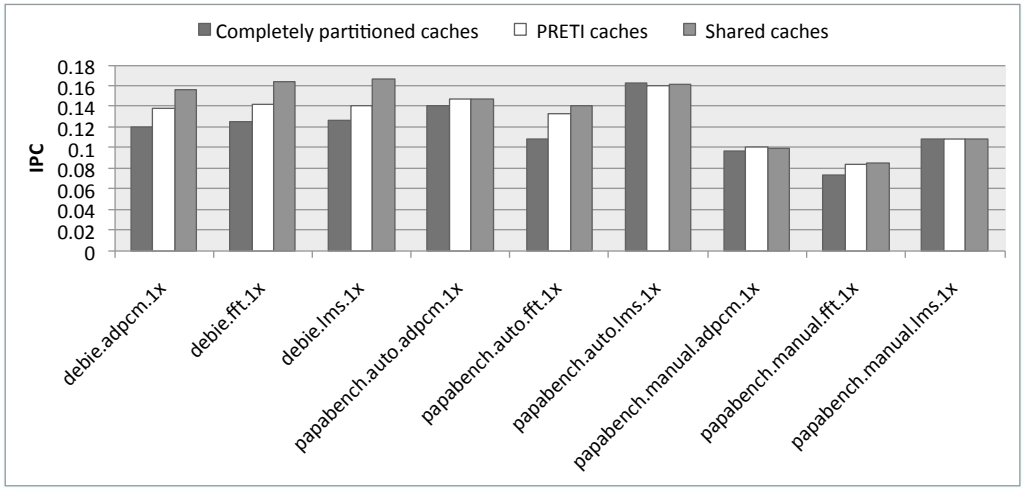
[17] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 300–303, New York, NY, USA, 2008. ACM.

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

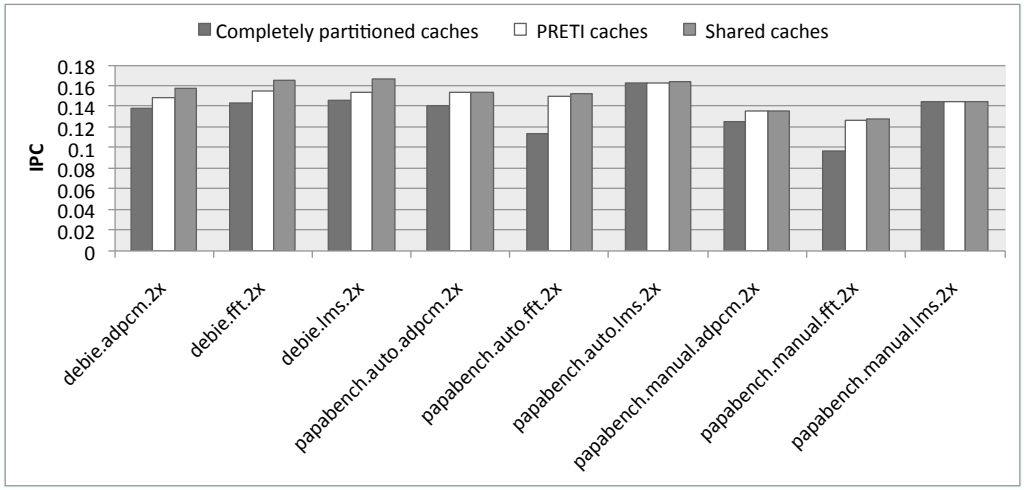
[19] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. 18(2-3):157–179, 2000.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem : overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

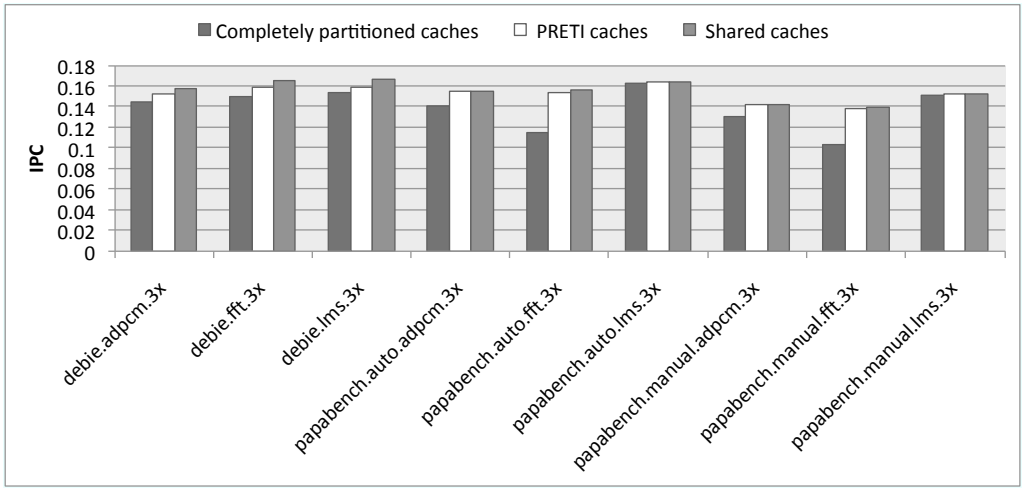
[21] A. Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2:315–327, March 1994.



(a) Performances of non-critical tasks under different configurations, at frequency 1x.



(b) Performances of non-critical tasks under different configurations, at frequency 2x.



(c) Performances of non-critical tasks under different configurations, at frequency 3x.

Figure 4: IPC of each non-critical tasks executed as the sole non-critical task, concurrently with a real-time workload, per processor frequency (1x, 2x or 3x).