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

Benjamin Lesage
University of Rennes 1, UEB, IRISA
Rennes, France
Benjamin.Lesage@irisa.fr

Isabelle Puaut
University of Rennes 1, UEB, IRISA
Rennes, France
Isabelle.Puaut@irisa.fr

André Seznec
INRIA Rennes - Bretagne Atlantique
Rennes, France
Andre.Seznec@inria.fr

ABSTRACT

Multithreaded processors, in the context of real-time systems, create the opportunity to mix, on the same hardware platform, the execution of a complex real-time workload and the execution of non-critical applications. But resources sharing, inherent to multithreading, hinders the timing analysis of concurrent tasks. Such analyses are critical to real-time tasks which have timing deadlines that must be met and enforced.

In this paper, we present the PRETI, Partitioned REal-TIme shared cache scheme, a flexible, low implementation-overhead, shared cache partitioning scheme. PRETI can preclude inter-task conflicts on shared caches, and their pessimistic impact on timing estimates, by allocating private cache space to real-time tasks. Therefore, uniprocessor, i.e. unithread, worst-case execution time (WCET) estimation techniques can be used. The remaining cache space, not reserved for currently running real-time tasks, is shared by all tasks running on the processor, in particular the non-critical ones, enabling high performances for these tasks.

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

Categories and Subject Descriptors

C.3 [Special-purpose and application-based systems]: Real-time and embedded systems; B.3 [Memory structures]: Design styles—Cache memories

1. INTRODUCTION

Need for performance, as well as the economic pressure to use off-the-shelf components or standard intellectual properties, 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 many application domains as possible.

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 for all inputs, the effective execution time will always be smaller or equal than the estimated WCET. Although research on WCET estimation on uniprocessor featuring instruction level parallelism and complex memory hierarchy has made important progress in the last two decades [28], 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. In the context of general purpose systems, many schemes have been proposed to improve tasks’ overall performance [10, 31, 22]. These best-effort strategies however prevent from offering guarantees in terms of available cache space to real-time tasks.

Yet 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 modifications 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 [14]. 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 allows the derivation of guarantees on the worst-case number of cache misses suffered by real-time tasks, using state-of-the-art static analysis methods originally defined for non-shared caches. 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 the conventional LRU replacement policy, known for its predictability. Furthermore, the benefits of similar modifications, to serve quality of service requirements for soft real-time tasks, have been demonstrated in previous works [18].

Hard real-time systems, in which safety is critical, are considered to demonstrate the interest of 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 and cache partitioning 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

Static cache analysis for real-time systems

Caches, although playing a key role for enhancing performance, raise predictability issues for real-time applications, for which execution times have to be upper-bounded a priori. Methods to provide upper bounds of the number of cache misses, operating through static program analysis, have been defined for several cache structures and cache replacement policies (e.g., [4, 7, 23, 29, 28]). Such methods were shown to provide both safe and tight estimates, provided that the replacement policy is predictable. A study of the predictability of cache replacement policies, presented in [23] has demonstrated that a strict LRU replacement policy allows for much tighter timing estimates than LRU approximations. For instance, the authors of [23] have shown that the analysis of a pseudo-LRU k-way cache is equivalent to the analysis of a LRU cache with an associativity degree of \( \log_2(k) + 1 \). Although the PRETI cache could be transposed to non-LRU replacement policies, LRU replacement policy will be used as a basis, for its predictability properties.

As far as shared caches are concerned, cache sharing on real-time systems has essentially been considered for multi-cores or multiprocessors, or in the context of time-shared uniprocessors. To the best of our knowledge, no specific studies has addressed sharing caches in multithreaded processors for real-time applications.

For multi-core architectures, state-of-the-art shared cache analyses [15, 6], stemming from abstract interpretation based analyses [4], have raised the problem of inter-task conflicts and their negative impact on tasks’ predictability. The use of bypass [6], information about tasks’ lifetimes [15], or the conjoint use of partitioning and locking [25] 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 for real-time systems

As already mentioned in the introduction, cache pseudo-partitioning has been considered for overall performance [10, 31, 22]. However, these general-purpose solutions do not fit real-time issues: predictability requirements.

In the context of real-time systems, cache partitionings have been explored mostly for preemptive systems, to eliminate cache related preemption delays [26, 30, 16]. 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 [12, 25].

The partitioning schemes introduced, or assumed, in related work [30, 16, 12, 25] 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 thus allows for better average-case performance than strictly partitioned caches.

Prioritized caches [26] 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 non-critical tasks. We use a similar on-demand cache line allocation mechanism. However, 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 been proposed for preemptive real-time systems [30, 16]. 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 [12] addresses the additional issue of shared data structures. Using this hybrid hardware and soft-
ware 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 [30, 16] and S.M.A.R.T. caching [12], allows for fine-grained partitions, but does not allow reclamation of disused resources within partitions. 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.

Modifications of the cache replacement policy, similar to the ones PRETI rely on, were proposed in previous works [22, 18, 11]. However, they were mostly studied from the average case performance [22], fairness [11] or Quality of Service [18] point of view. In contrast, this study focuses on shared cache behaviour’s predictability, for hard real-time systems, and analysability. As an example, using the Virtual Private Caches capacity manager [18], which is the closest replacement policy and objective-wise to PRETI, multiple blocks may be suitable candidates for eviction, and an additional fairness policy is required to select the evicted one. Using PRETI, such an ambiguity is not present; when eviction is required, only one block is deemed as a suitable candidate.

Cache management and mixed-criticality real-time systems
To the best of our knowledge, no specific study has addressed the issue of cache management for mixed-criticality real-time systems. The PRETI cache scheme has been defined for these real-time systems mixing real-time tasks and non-critical tasks.

Furthermore, PRETI caches can be integrated with other mechanisms tackling the predictability issue for real-time systems, in different levels of the memory hierarchy. Such works targeting the bus arbitration policy [24, 19] or the memory controller [21] allow for upper-bounding access latencies to the different layers of the memory hierarchy.

3. THE PRETI CACHE 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 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 analyses, 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 of 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 blocks most recently 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. 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) least recent cache lines used by a task beyond its private space’s allocated capacity.

Figure 1 illustrates the 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 real-time tasks, Task 1 and Task 2. Task 1 has been allocated two cache ways, whereas Task 2 can claim the exclusive ownership of one cache 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 then composed of two cache lines, the unallocated one and one line unclaimed by Task 1 at this time instant.

This example 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, if not empty,
- 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. Therefore this block enters the private space of the miss-triggering task. The inserted block may also push the least recently used block of this private space to the shared space, to keep said private space’s size within its allocated capacity.

By construction, 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; only the owner of a private space can evict blocks belonging to this space, and only if said block is the least recently used one in the private space and no candidate block belongs to the shared space.

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. Upon a hit, the LRU tags must be modified as on a conventional LRU cache and the access hit time is not modified.

Upon a miss, the selection of the replaced block is different from the one in a conventional LRU cache. An extra information must be associated with the cache block, the task id of the owner task, updated on each 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. In addition, the cache must hold, for each possible task id, its allocated number of lines.

![Figure 2: Storage requirements for PRETI caches: each cache line is extended with a task id.](image)

The cost, in terms of storage requirements, of the proposed implementation is low: \( \lceil \log_2(T + 1) \rceil \times \ell \) bits for the task ids and \( \lceil \log_2(a + 1) \rceil \times T \) bits to hold the maximum partition sizes, expressed in number of cache ways than can be allocated to each task id, where \( \ell \) is the number of cache lines, \( T \) the maximum number of distinct private spaces which may reside in the cache at the same time, and \( a \) is the cache associativity.

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. The oldest cache line which task id is either unset or invalid (used to belong to a completed task), or refers to a task id with over-allocated lines in this set, is selected. task id invalidation happens upon each task’s completion so that its private space cache line reenters the shared space.

3.3 Analyzing PRETI cache’s behavior for a real-time task

Performance guarantees are required for real-time tasks. By construction, if a N-ways partition is reserved for a task, the PRETI cache ensures that each set in the shared cache maintains the task’s N most recently used blocks. The private partition can be seen as a small private cache, implementing LRU replacement and free of inter-task conflicts. As a consequence, state-of-the-art static cache analysis techniques, originally designed for private caches [27, 7, 13] can be directly used. With respect to WCET analysis, PRETI and strictly-partitioned caches behave the same way.

Using a PRETI cache, the estimation of a task’s WCET is independent of the underlying scheduling algorithm used by the system. This is opposed to the use of a shared cache where tasks’ lifetime, and therefore scheduling, provide insightful information on potential conflicts, requiring iterative analyses to improve tasks’ WCET estimates [15]. Yet, upon system validation, one has to ensure that, anytime, the sum of space allocated to concurrently running tasks remains smaller than the cache size. This property is implicitly validated for joint partitioning-scheduling algorithms [20]. Nevertheless, it should be noted that sharing data among real-time 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 [15, 6] 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. Possible solutions in the context of PRETI cache would be to reserve a private partition for shared data, or to reserve private partitions to non-shared data.

4. EXPERIMENTAL RESULTS

In this section, after a presentation of the experimental setup (§ 4.1), we demonstrate 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

**Hardware platform.**

For the purpose of our experiments, we will consider a very simple model of time-multi threaded 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. As mentioned in the introduction, the contribution of the memory hierarchy to the WCET is very large (tens of cycles per instruction) and plainly justify our simple processor model which essentially ignores the contributions of instructions dependencies and pipeline hazards.

The processor features separate instruction and data caches, both of 4KB and 8-way associative, 32 bytes cache blocks are used for both caches. The replacement policy is always the same for both caches, but may depend on the configuration: shared LRU, strictly partitioned LRU, or PRETI. Upon a miss, the miss penalty of 150 cycles is observed by the thread, but the memory can service up to 6 misses con-
currently, 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.

Codes are compiled into MIPS R2000/R3000 code using gcc 4.1 without any code optimization and using the default linker memory layout.

**Benchmarks.**

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 [5].

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 [8], developed by Space Systems Finland, monitors the impact of space debris and micro-meteoroids using electrical and mechanical sensors. The Papabench tasks [17] 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 scheduling.**

Since real-time tasks have to meet their deadline, a real-time scheduling policy is used to schedule them: NP-EDF [9] (non-preemptive Earliest Deadline First). Upon each release of a real-time 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 use of a non non-preemptive scheduling for critical tasks eliminates the impact of cache-related preemption delays on the performance of real-time tasks. We focus on the impact of PRETI on inter-task conflicts which stem from cache sharing because of the multi-thread processor, not because of preemptions.

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

Each real-time task is statically allocated to a single hardware thread. In case there are more real-time tasks than hardware threads, multiple tasks may be mapped to the same hardware thread, and have the same size of private cache partition. The task-to-thread mapping algorithm, as well as the determination of partition sizes, are detailed in section 4.2.

**Evaluation of schedulability of real-time tasks.**

Real-time tasks have to meet their deadlines in all situations, including the worst-case. This implies that all real-time tasks have to be known a priori, as well as their timing properties (periods, deadlines, worst-case execution times). A schedulability test is then executed before running the applications to verify the system schedulability.

A standard schedulability test for periodic tasks scheduled under NP-EDF is used [9]. According to the selected schedulability test, a set of periodic tasks with deadlines equal to their periods is schedulable if and only if 1) the thread is not overloaded; the cumulated utilization, WCET over period, of the tasks on the thread cannot exceed one, and 2) for each time interval between the smallest and the biggest periods, the cumulated demand, in WCET, of all tasks’ instances that can be scheduled on the interval must be less or equal to the length of the interval. This is a sufficient and necessary condition [9].

As far as WCET estimation of real-time tasks is concerned, a state-of-the-art static WCET estimation tool is used [3]. The tool statically analyzes the tasks’ code to obtain addresses of referenced code and data. Memory references are then statically classified as hits/misses using the most recent static cache analysis techniques presented in [7, 2]; every reference is classified as hit if the block is for sure in the cache, and as a miss otherwise. The worst-case execution time of each task is then computed using the standard Implicit Path Enumeration Technique (IPET) technique for WCET computation [28], that computes the longest execu-

<table>
<thead>
<tr>
<th>Task set</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>debie</td>
<td>acquisition_task, hit_trigger_handler, monitoring_task, tc_execution_task, tc_interrupt_handler, tm_interrupt_handler</td>
</tr>
<tr>
<td>papabench.manual</td>
<td>tasks 1, 2, 3, 4, 5, 6, 7, 8, and 13</td>
</tr>
<tr>
<td>papabench.auto</td>
<td>tasks 3, 4, 5, 7, 8, 9, 10, 11, 12, and 13</td>
</tr>
</tbody>
</table>

Table 2: Task sets

Three different sets of periodic real-time 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 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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Code (bytes)</th>
<th>Data (bytes)</th>
<th>Bss (bytes)</th>
<th>Stack (bytes)</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debie</td>
<td>acquisition_task</td>
<td>5968</td>
<td>1280</td>
<td>101904</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>hit_trigger_handler</td>
<td>2480</td>
<td>647</td>
<td>218</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>monitoring_task</td>
<td>12372</td>
<td>567</td>
<td>65772</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>tc_execution_task</td>
<td>12928</td>
<td>1280</td>
<td>101904</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>tc_interrupt_handler</td>
<td>872</td>
<td>23</td>
<td>168</td>
<td>56</td>
</tr>
<tr>
<td>Papabench</td>
<td>task 1</td>
<td>2000</td>
<td>74</td>
<td>74</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>task 2</td>
<td>696</td>
<td>52</td>
<td>242</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 3</td>
<td>1492</td>
<td>64</td>
<td>66</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 4</td>
<td>372</td>
<td>4532</td>
<td>568</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 5</td>
<td>1440</td>
<td>2357</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 6</td>
<td>8508</td>
<td>572</td>
<td>302</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>task 7</td>
<td>1152</td>
<td>194</td>
<td>158</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 8</td>
<td>264</td>
<td>312</td>
<td>46</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>task 9</td>
<td>10284</td>
<td>552</td>
<td>511</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>task 10</td>
<td>33700</td>
<td>8192</td>
<td>302</td>
<td>1160</td>
</tr>
<tr>
<td></td>
<td>task 11</td>
<td>268</td>
<td>90</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 12</td>
<td>1120</td>
<td>145</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>task 13</td>
<td>21716</td>
<td>284</td>
<td>255</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Malfunction</th>
<th>Code (bytes)</th>
<th>Data (bytes)</th>
<th>Bss (bytes)</th>
<th>Stack (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpem</td>
<td>7872</td>
<td>1390</td>
<td>436</td>
<td>152</td>
</tr>
<tr>
<td>hit</td>
<td>3772</td>
<td>88</td>
<td>128</td>
<td>296</td>
</tr>
<tr>
<td>lms</td>
<td>2992</td>
<td>248</td>
<td>1092</td>
<td>184</td>
</tr>
</tbody>
</table>

Table 1: Benchmark characteristics
4.2 Impact of cache partitioning on schedulability

In this section, we show the benefits of using the PRETI cache, in comparison to a shared cache, for the schedulability of real-time tasks with tight timing constraints.

For each real-time workload, we determine the minimum number of processor cycles per second of the system, the frequency, for which the real-time workload is schedulable. By schedulable, we mean that all real-time tasks meet their deadline even if all of them execute during their WCET estimate.

To compute this minimum frequency when using the PRETI cache, we systematically explore all the possible (cache partitioning, task allocation) pairs among the threads and, for each one, determine the minimum frequency ensuring schedulability under this configuration. This minimum frequency is computed by scaling the frequency up until the system is schedulable (based on tasks’ estimated WCETs in cycles under the configuration, their execution frequencies and the NP-EDF schedulability test). The best pair, that is the one minimising the frequency, is then selected. Note that, for the sake of simplicity, it is assumed that the main memory frequency scales along with the processor frequency, hence ensuring a constant 150 cycles miss latency. Were it not so, the WCET of each task would have to be estimated for each processor frequency while considering an increased memory latency in cycles as the processor frequency is scaled up.

Similarly, all the possible task allocations are explored for the shared cache to determine the minimum frequency for this configuration.

Assuming a fully shared cache leads to a minimum frequency for guaranteeing schedulability much higher (respectively 13x, 5.8x, and 5.6x for the debie, papabench.auto and papabench.manual workloads) than the minimum frequency needed for the PRETI cache. These results are reported in Table 3. We also illustrate for each hardware thread the complete execution, that is a hardware thread cannot complete an instruction 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.

4.3 Performance 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 for real-time tasks.

In order to illustrate this property, we run simulations assuming different combinations of processor frequencies and cache configurations. Considered processor frequencies are respectively 1, 2, and 3 times the minimum frequency necessary to schedule the real-time workload on the PRETI cache, as calculated in section 4.2. Similarly, the assumed task-to-thread mappings are the ones corresponding to the best configuration of the PRETI cache. The system is run for 1.5 billion cycles for papabench.auto and papabench.manual.

For each frequency, three different configurations are compared, the PRETI cache, a completely partitioned cache and a shared cache. This 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 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 task.

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.

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 evict the blocks from the other hardware threads. In our experiments, we allocate all the cache ways among the hardware threads (see Table 4). These partitions are compatible with the minimum requirements in Table 3; the remaining ways are distributed among the threads.

First round of simulation: A non-critical task is allocated to each hardware thread. When no critical task is required to run on a thread, this thread’s non-critical task execute repetitively. The allocated non-critical tasks are lms, adpcm and fft, respectively on each of the three hardware threads.

We measured the performance of each non-critical task in Instruction Per Cycle (IPC) over the whole simulation duration. The maximum achievable performance for the thread would be 1 instruction every 3 cycles if there were no cache miss and no interrupts by the critical tasks.

The results provided in Figure 3 present the normalized IPC of the non-critical tasks using either a completely partitioned cache or a PRETI cache. Their IPC using a shared cache is used as the baseline. They illustrate the benefits of sharing caches for the non-critical tasks. The results are provided for the debie, papabench.auto and papabench.manual, resp. in Figure 3a, 3b, and 3c. 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 normalized sum of the IPC of the non-critical tasks running on each hardware thread.

Two behaviors can be observed; non-critical tasks perform better either using the shared cache over the fully
**Table 3:** Best configurations for the debie, papabench.auto and papabench.manual under the best partitioning or using shared caches.

<table>
<thead>
<tr>
<th>TASK</th>
<th>Instruction cache partition</th>
<th>Data cache partition</th>
<th>Real-time tasks' instructions per second on the predicted critical path</th>
</tr>
</thead>
</table>
| **debie using partitioned caches**
(1.5GHz processor frequency) | 7 ways  | 4 ways | 54,109,012 |
| **debie using shared caches**
(19.5GHz processor frequency) | 1 way  | 1 way | 18,366,000 |
| **papabench.auto using partitioned caches**
(3.0Mhz processor frequency) | 4 ways  | 1 way | 90,031,033 |
| **papabench.auto using shared caches**
(17.5Mhz processor frequency) | 3 ways  | 3 ways | 60,031,033 |
| **papabench.manual using partitioned caches**
(3.1Mhz processor frequency) | 3 ways  | 1 way | 91,024 |
| **papabench.manual using shared caches**
(17.5Mhz processor frequency) | 3 ways  | 3 ways | 90,260 |

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

<table>
<thead>
<tr>
<th>TASK</th>
<th>Instruction cache partition</th>
<th>Data cache partition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>debie</strong></td>
<td>7 ways</td>
<td>4 ways</td>
</tr>
<tr>
<td><strong>papabench.auto</strong></td>
<td>3 ways</td>
<td>3 ways</td>
</tr>
<tr>
<td><strong>papabench.manual</strong></td>
<td>3 ways</td>
<td>3 ways</td>
</tr>
</tbody>
</table>

partitioned one, or the fully partitioned cache allows for marginally better performances than the shared cache. In all cases, the performances of the non-critical workloads using the PRETI cache are in-between their ones using a fully partitioned cache or a shared cache, and close to the best of the two. When always partitioning the cache, as exemplified in Figure 3a, starvation occurs for Thread 2 and, to a lesser extent,
Thread 1. The ways of the cache are mainly reserved to guarantee the safety on Thread 0. On debie, tasks running on Thread 2 have access to only one data cache way.

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). Focusing on Thread 1 under the debie workload (Figure 3a), 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 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 occurs on the two papabench workloads (see respectively Figure 3b, and Figure 3c for papabench.auto, and papabench.manual). 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 Thread 1 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 exhibits lower temporal locality. It is mitigated by the fact that the performance of the other tasks improve when sharing the cache space, that is using shared or PRETI caches.

**Second round of simulation.** We only execute a single instance of a non-critical task concurrently with the real-time workload. If not already running, this non-critical task is scheduled whenever a hardware thread becomes available. We run 3 sets of simulations using respectively lms, adpcm and fft as this single non-critical task. Simulation results are presented in Figure 4. The normalized IPC of the non-critical task is presented, from top to bottom, by processor frequency (resp. Figure 4a, 4b, and 4c for frequencies 1x, 2x, and 3x) then, from left to right, by concurrent real-time workload, and finally, per workload, for each non-critical task and cache configuration. Again, the IPC under the shared cache scenario is used as the baseline.

The results globally illustrate that the PRETI cache structure allows the non-critical task to benefit from shared cache space. Completely partitioning the cache results in a worse cache behavior than using a fully shared cache, especially at higher frequencies, 2x and 3x. Considering PRETI caches, and as stated earlier, the overall cache is completely shared for more than 77% of the time for each simulation. Therefore, as frequencies increases, 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 cache shrinks.

As was the case in the previous round of simulations, 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 (see Figure 4a, sixth data set), Thread 0 is executing real-time tasks 64% of the time. Therefore, at least 64% of the time, lms has to share cache space with others, which again induces a slight performance degradation using shared or PRETI caches over the use of a 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, nowa-
days, 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 intellectual properties 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 volumes, or if the modification is very marginal and does not impair the rest of the design. This applies to 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 in 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 has resulted in a larger gap between actual performance and predicted worst-case performance [7]. This has lead some authors to oppose to completely avoid cache sharing, and instead rely on only private caches or, equivalently, strict partitioning schemes [20]. 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 multi-threaded 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 analyses 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 close 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.

6. REFERENCES


