ABSTRACT
Multi-core architectures are well suited to fulfill the increasing performance requirements of embedded real-time systems. However, such systems also require the capacity to estimate the timing behavior of their critical components. Interference between tasks, as they occur on standard multi-core micro-architectures due to cache sharing are still difficult to predict accurately. An alternative is to remove these indirect interferences between tasks through partitioning of the shared cache and through the use of partitioned task scheduling.

In this paper, we present a new algorithm for joint task and cache partitioning in multi-core systems scheduled using non-preemptive EDF. The main novelty of the algorithm is to take into account the tasks’ period repartition in the task partitioning problem, which is critical in a non-preemptive context. Other task properties such as task cache requirements are also considered to optimize cache partitioning. Experiments show that our algorithm outperforms the state-of-the-art algorithm for tasks and cache partitioning, named IA3 [18], in terms of schedulability, specially when tasks have varied periods.

1. INTRODUCTION
The need for performance is one of the major factors leading to the advent of multi-core architectures in embedded systems. Such architectures also have the added benefit of being power-efficient. In the context of real-time systems, time-critical tasks must be guaranteed to meet their deadlines in all cases, even the worst. Worst-Case Execution Time (WCET) estimation methods [21] are used as a basis to estimate tasks’ timing behavior and validate systems.

One of the issues raised by multi-core architectures in the context of real-time systems is the sharing of hardware resources between cores. For example, it is common that the last level cache is shared, resulting in hard-to-predict interferences between tasks running on different cores. One solution to this issue is to partition the last level cache among cores. So far, most cache partitioning methods have been proposed for single core architectures.

Task scheduling on multi-core architecture, in addition to distributing the tasks in time, have to allocate the tasks to the cores. Scheduling strategies can be divided in two broad categories [8]. Global scheduling strategies allow tasks to be scheduled on any of the cores. In contrast, partitioned scheduling strategies assign each task to a single core and forbids task migrations between cores and the resulting overhead. As an added benefit, partitioned scheduling allows to reuse mono-core schedulability conditions.

Most existing task partitioning algorithms assume a single WCET per task, and thus assumes that the execution context of a task (e.g. allocated number of partitions when cache partitioning is used) is a priori known. To the best of our knowledge, only one research study [18] jointly considers cache partitioning among cores and task partitioning for multi-core platforms. In [18], an algorithm, called IA3, for Interference Aware Allocation Algorithm, jointly partitions tasks and cache for non-preemptive EDF scheduling. IA3 takes into account multiple execution contexts for every task, in particular, different cache partition sizes. However, it does not take advantage of all properties of tasks in the partitioning process, in particular the repartition of tasks’ periods, which is crucial when using non-preemptive scheduling.

In this work we present PDPA, for Period Driven Partitioning Algorithm, a joint task and cache partitioning algorithm for multi-core systems scheduled using non-preemptive Earliest Deadline First (NP-EDF). On the one hand, PDPA takes advantage of relationships between tasks periods for partitioning tasks among cores. On the other hand, tasks cache usage is accounted for when partitioning the shared cache among cores. Experimental results show that PDPA outperforms IA3 in terms of percentage of schedulable task sets in most situations.

The reminder of the paper is organized as follows. Related work is presented in section 2. Notations and problem
formulation are given respectively in sections 3 and 4. Our algorithm, PDPA, is described in section 5. Experimental results are given in section 6. A summary of the contributions and directions for future work are finally given in section 7.

2. RELATED WORK

Real-time scheduling strategies for multi-cores architectures can be divided into two broad categories: global and partitioned [8] scheduling. Partitioned scheduling assigns each task to a single core and prevents tasks migrations between cores. The tasks assigned to each core can then be scheduled using state-of-the-art mono-core scheduling algorithms, and the well established corresponding schedulability tests can be used unmodified. In contrast, global scheduling strategies allow task migrations, at different levels: either at any time during the execution of job, or between the execution of two jobs of the same task. Task migrations in systems with caches introduce a delay to refill the cache hierarchy after a migration, which is hard to upper-bound. Therefore, in this paper, we will focus on partitioned scheduling.

The tasks partitioning problem is similar to the bin packing problem, known to be NP-hard. The classical heuristics defined so far for solving the bin packing problem can be used (first-fit, best-fit, worst-fit) [7, 4]. These heuristics assume a single WCET per task; thus, when used on architectures with shared caches, they assume for each task the worst interference from tasks running on the other cores. In other terms, they do not fully take benefit of the hardware context. In contrast, our algorithm considers one WCET per task and cache partition size and thus is aware of the task hardware context. Tasks and cache are partitioned jointly such that hardware resources (shared cache and cores) are used as efficiently as possible.

There are many methods for WCET calculation which take caches into account [17, 11, 20, 5]. They have been recently extended to multi-core platforms with shared last level caches [14, 10]. However, those methods tend to be pessimistic when pressure on the shared cache is high. Rather than accounting for inter-task interferences during WCET estimation, cache partitioning methods eliminate such interferences. The shared cache is partitioned (among cores or tasks), and a core (respectively, a task) can only access its allocated partition, thus removing all interferences caused by the shared cache. On the downside, every core/task can only use a subset of a cache, which may result in decreased performance. In this paper, we will focus on cache partitioning. Different partitioning strategies exist [3, 19]. In practice, partitioning can be software-implemented [15], or rely on hardware support [13]. Our approach applies to both hardware and software-implemented partitioning schemes.

A related approach for tasks and cache partitioning is presented in [18]. In [18], an algorithm called IA3 jointly partitions tasks on cores and caches among cores. IA3 considers multiple WCETs per core depending on the run-time environment of the task (number of tasks running on the other cores and competing for bus access, and amount of shared cache available). Similarly to our proposal, IA3 takes tasks utilizations and cache usage into account into the partitioning process. Our algorithm has the same objective and scheduling strategy than IA3 (NP-EDF). However, our algorithm does not only consider tasks' utilizations and memory requirements, but also considers relationships between tasks periods, making it much more effective than IA3 in many situations, in particular when task periods are very different.

3. SYSTEM MODEL AND NOTATIONS

3.1 Architecture

We consider a real-time system composed of $N$ tasks to be executed on a multi-core architecture composed of $M$ identical cores. Each core may have private caches (e.g. private and separate L1 instruction and data caches). The architecture has a shared last level cache of size $S$, which can be divided into $K$ equal size partitions. The way cache partitioning is implemented (using software-based or hardware-based techniques) is not important for the cache and task partitioning algorithm, as far as each core can be allocated any number of partitions between $0$ and $K$. Similarly, the characteristics of the cache hierarchy (number of levels of private caches, associativity, replacement policy) can be disregarded to the extent that there is a single level of shared cache and one is able to compute WCETs for any number of partitions in the shared cache.

3.2 Task properties and task scheduling

Each task $t_i, 1 \leq i \leq N$ is periodic, with a fixed period $P_i$. Without loss of generality, we assume $(i < j) \Rightarrow (P_i \leq P_j)$. We consider a system with implicit deadlines $D_i$ ($D_i = P_i$). The other characteristics of tasks, namely WCET and associated utilization, depend of the amount of cache that the task will dispose of, and are defined below.

**Tasks WCETs**

Each task has a different WCET depending on the number of cache partitions allocated to the core it will be running on. The WCET of a task $t_i$ when the task disposes of $n$ partitions of the shared cache will be noted $C_i^n$. When the number of cache partitions is obvious from the context, notation $C_i$ will be used as a shortcut for $C_i^n$.

**Reference utilization and WCET**

Although there are multiple WCETs for each task, it is convenient for our cache partitioning algorithm to dispose of a single reference WCET to guide the task partitioning algorithm. Although the shapes of the WCET functions differ depending on the tasks memory access patterns, there are common characteristics between them. As an illustration, Figure 1 represents the WCET function of task $fft$ from the Mälardalen benchmark suite [16]. The WCET has been computed with the Heptane WCET estimation tool [6], assuming a two-level cache hierarchy with a unified L2 cache.
The general tendency of the WCET function is a decrease of the task WCET when the number of partitions allocated to the task in the shared cache increases. Another common point is a large difference between $C^0_i$ and $C^1_i$. In the following, value $C^1_i$ will be used by the task allocation algorithm and will be termed reference WCET of task $t_i$. $C^1_i$ is selected as a reference WCET instead of $C^0_i$ because of the large difference between $C^1_i$ and $C^0_i$ and the fact that most cores will have (at least) one cache partition. The corresponding processor utilization

$$U(t_i) = \frac{C^1_i}{P_i}$$

will be termed reference utilization.

### Optimal number of partitions

Another property for tasks with a sufficiently small working set is that beyond a given number of cache partitions, the WCET does not decrease anymore. The optimal number of partitions for a task $t_i$, noted $\kappa_i$, is defined as follows:

$$\kappa_i = \min\{k \in [0, K] | \forall k' > k, C^k_i \leq C^{k'}_i \}$$

In our example the optimal number of partitions for the considered task is 5. For tasks with a working set larger than the capacity of the last level cache, $\kappa_i = K$.

### Variability

It is important for the cache partitioning algorithm to determine if the WCET of a task varies much or little when allocating more cache to it. The variations of WCET can be very different depending on the nature of the task. This is captured through the concept of variability of a task $t_i$, noted $V(t_i)$ defined the absolute value of the slope of the line between its reference WCET and its optimal WCET (the dotted line in Figure 1). Formally:

$$V(t_i) = \left| \frac{C^{\kappa_i}_i - C^1_i}{\kappa_i - 1} \right|$$

The introduced concept of variability as defined in this paper is very close to the one of WCET-sensitivity as defined in [18].

### Task scheduling

In this article we focus on systems scheduled using non-preemptive Earliest Deadline First (NP-EDF) [12] on each core. Non-preemptive scheduling was selected because it eliminates cache-related preemption delays (CRPDs), and thus alleviates the need for complex and still pessimistic CRPD estimation methods.

## 4. Problem FORMULATION

The considered joint tasks and cache partitioning problem consists in assigning each task to one core and partitioning the last level cache among cores, such that the resulting system is schedulable under non-preemptive EDF.

A system with a fixed number of cores $M$, equipped with shared cache that can be partitioned in $K$ partitions is assumed. The problem consists, for a task set with $N$ tasks, in producing a configuration that is both valid and schedulable. A configuration is a set of $M$ pairs $(\tau_c, k_c)$, $1 \leq c \leq M$, where $\tau_c$ is the set of tasks assigned to core $c$ and $k_c$ is the number of partitions of the last level cache assigned to $c$.

- A configuration is valid if (i) it maps every task to one and only one core (ii) each core has a number of partitions assigned, and (iii) the total number of cache partitions assigned to the cores is less than or equal to the total number of partitions $K$.

- A valid configuration is schedulable if, on every core $c$, the set $\tau_c$ of tasks assigned to the core is schedulable under non-preemptive EDF. In a valid configuration, each core $c$ has $k_c$ partitions reserved, and therefore, the WCET of each task $t_i$ is known and equal to $C^C_{i,c}$. Configuration for core $c$ $(\tau_c, k_c)$, is schedulable under NP-EDF if it satisfies the two conditions defined in [12]:

$$\sum_{t_i \in \tau_c} \frac{C^C_{i,c}}{P_i} \leq 1 \quad (1)$$

$$\forall t_i \in \tau_c, \forall L, P_1 < L < P_i : L \geq C^C_{i,c} + \sum_{j=1}^{i-1} \left| \frac{L-1}{P_j} \right| C^C_{j,c} \quad (2)$$

## 5. The PDPA Algorithm

PDPA, for Period Driven Partitioning Algorithm, is a new algorithm for joint task and cache partitioning that takes benefit of the repartition of tasks periods in the partitioning process. Paragraph 5.1 gives the basic principles of the algorithm, which is fully detailed in paragraph 5.2.

### 5.1 Algorithm design rationale

Equation (1) in the schedulability conditions of NP-EDF given above clearly shows that the schedulability of tasks on
each core is influenced by the utilizations of tasks mapped to that core. Equation (2) shows that tasks periods also have an impact. Unfortunately, equation (2) does not sufficiently explicit which relations between task parameters help satisfying the condition. Instead of directly using equation (2) in the task partitioning process, PDPA uses equation (3) transposed from [1, 9] to guide task partitioning (recall that tasks are ordered by non decreasing periods):

$$\forall t_i \in \tau, C_i + \sum_{j=1}^{i-1} \left( C_j + \frac{C_i}{P_f} (P_i - P_j) \right) \leq P_i$$  \hspace{1cm} (3)

The important part of the equation is the following term:

$$\frac{C_i}{P_f} (P_i - P_j)$$

We call this the relative weight of the task $t_j$ for the task $t_i$. With condition (3) we see that the aim for each task $t_i$, is to minimize the sum of the relative weights of $t_i$ for the tasks with a period lower than the one of $t_i$. We also see that the larger the period of $t_i$ is, the bigger this sum can be.

Based on these considerations, PDPA will initially place tasks to cores depending of their periods and utilizations, and guided by the introduced concept of relative weight. In the second phase, PDPA will re-consider the assignment of some tasks to make the whole system schedulable, by taking into account the tasks memory requirements.

5.1.1 Phase 1. Initial placement

According to equation (3), the computation of the sum of relative weights for a task $t_i$ considers only the tasks with a period lower than $P_i$. Therefore, if a task has a high utilization and the biggest period of the core, it is beneficial for schedulability since the task utilization will only be considered once. In addition, on each core, in order to meet equation (3), we want the difference between tasks periods to be minimized, or that the tasks’ utilizations are small. The tasks that have their periods far from the task with the higher period must have a small utilization.

Based on these considerations, PDPA will initially place some tasks, called critical tasks. Critical tasks (see paragraph 5.2 for details) are selected among the tasks having a big reference utilization and a low variability. This criterion is defined such that for all cache partitionings, those tasks will always have a big utilization. In order to minimize differences between task periods and satisfy equation (3), critical tasks are also selected in such a manner that their periods are distributed among the period range of the task set. $M$ critical tasks are selected and each of them is placed on a core, as depicted in Figure 2.

Once the critical tasks are selected and assigned to a core, the other tasks are placed according to their utilization and period, in order to minimize the difference of task periods of equation (3). A task of period $P_i$ is assigned to a core which has a critical task of period greater than $P_i$. On Figure 2, the period of critical tasks is marked with a vertical dotted line; graphically, each non-critical task has to be at the left of a critical task.

5.1.2 Phase 2. Modification of initial placement

The initial placement may result in cores with many tasks which will not be schedulable, and cores with very few tasks. Moreover, tasks memory requirements are not yet taken into account. Tasks with a close optimal number of partitions should ideally be on the same core. When this is not possible, variability is considered to give priority to tasks with a large variability.

The second phase of PDPA consists in re-assigning tasks to cores. Tasks from unschedulable cores will be assigned to other cores taking into account their properties (processor utilization, cache requirements and variability).

5.2 The algorithm in details

The main phases of the PDPA algorithm are presented in algorithm 1. PDPA takes as a parameter the task set $\tau$. The output of PDPA is a valid and schedulable configuration $\phi$ if the algorithm found one, or an empty set otherwise. PDPA is divided in three phases, each of them to be detailed later in this paragraph.

The first phase (lines 2 to 7) consists in obtaining an initial task placement. First, the critical tasks are determined, assigned to the cores, and then removed from the task set in function PLACE_CRITICAL_TASKS. Variable $\phi'$ is the configuration with the critical tasks assigned, and $\tau'$ the task set without the critical tasks. Those two sets are fixed and not modified afterwards. Then, the other non-critical tasks are placed (call to function PLACE_OTHER_TASKS). Then, the ASSIGN_CACHE_MIN function is called. The aim of the ASSIGN_CACHE_MIN function is, for every core $c$, to determine how many cache partitions must be assigned to $c$ such that the tasks assigned to $c$ are schedulable, when possible. At this step of the PDPA algorithm, the obtained configuration might be neither valid nor schedulable.

During the second phase (line 8 to 13), while the current configuration is not schedulable or is not valid, the algo-
The first phase of the algorithm is to identify critical tasks. Identification and placement of critical tasks returns the empty set.

As discussed earlier, periods of critical tasks have to be as far as possible from each other. The periods of critical tasks have to be as minimum as possible from each other.

The initial value of the threshold is 0.9, meaning the task has to be among the 10% of tasks with the lower variability. As discussed earlier, periods of critical tasks have to be as far as possible from each other. If the selected task has a period distant of at least δ from an already placed critical task, it is allocated to a new core (lines 10 and 11). This process is repeated until there are no more tasks in the task set. If all the cores have a critical task assigned, then the function returns. Else, the threshold is decreased (i.e. tasks with a lower variability are considered as candidate critical tasks) and the whole process is repeated with the remaining tasks.

Once the critical tasks are assigned, the configuration is stored in φ’. φ’ contains exactly one critical task per core. Critical tasks in φ’ will never be assigned to another core after the initial task placement, contrary to non-critical tasks.

### Placement of non-critical tasks

The initial placement of non-critical tasks is performed by function PLACE_OTHER_TASKS. A non-critical task τ is placed on the core that hosts the critical task τ, with the smallest period greater or equal to Pt. More formally, Pt is placed on core c, hosting critical task tc, if:

\[ P_t \leq P_c \]

and

\[ | P_t - P_c | \]

is minimized.

The result of task placement on our example is depicted in figure 2.

### Assignment of number of cache partitions

Function ASSIGN_CACHE_MIN assigns to each core c the minimum number of cache partitions such that the tasks assigned to c are schedulable. For each core, function ASSIGN_CACHE_MIN sets the number of cache partitions to 0,

- for a critical task
- for a non-critical task
- for a period distant of at least δ from an already placed critical task.

## Algorithm 1

```plaintext
function PDPA
  φ ← ∅
  i ← 0
  // Phase 1. Initial task placement
  (φ’, τ’) ← PLACE_CRITICAL_TASKS(τ)
  φ ← PLACE_OTHER_TASKS(τ’, φ’)
  φ ← ASSIGN_CACHE_MIN(φ)
  // Phase 2. Task re-assignment until schedulable
  while ¬ (SCHEDULED(φ) ∧ VALID(φ)) ∧ (i < nbTry)
    φ ← MOVE_TASKS(τ’, φ’, φ)
    φ ← ASSIGN_CACHE_MIN(φ)
    i ← i + 1
  end while
  // Phase 3. Finalization of cache partitioning
  if SCHEDULED(φ) ∧ VALID(φ) then
    φ ← ASSIGN_TOTAL_CACHE(φ)
  end if
  return φ
end function
```

## Algorithm 2

```plaintext
function PLACE_CRITICAL_TASKS(τ)
  φ ← ∅
  τ’ ← τ
  threshold ← 0.9
  while card(φ) ≤ M ∧ threshold > 0 do
    for all t ∈ τ’ by decreasing utilization do
      v ← NORMALIZED_VARIABILITY_RANK(t, τ)
      if v > threshold then
        τ’ ← τ’ − {t}
        φ ← φ ∪ {t, 0}
      end if
    end for
    threshold ← threshold − 0.1
  end while
  return (φ, τ’)
end function
```

Algorithm 1: Identification and placement of critical tasks

Algorithm 2: Placement of number of cache partitions
and increments it until either equations (1) and (2) are satisfied or the number of cache partitions is equal to $K + 1$. If the sum of the number of partitions assigned to the cores is less than or equal to $K$ after calling `assign_cache_min`, the configuration is schedulable.

### 5.2.2 Phase 2. Modification of initial placement

Function `move_tasks` modifies the assignment of non-critical tasks. Tasks to be re-assigned are selected among non schedulable cores. By construction of function `assign_cache_min`, a core $c$ is not schedulable if $k_c > K$. Function `move_tasks` is presented in Algorithm 3.

**Algorithm 3**

```
1: function move_tasks($\tau', \phi', \phi$)
2:     for all $c = (\tau_c, k_c) \in \phi$ do
3:         if $k_c > K$ then
4:             $t \leftarrow$ MAX_SCORE_TASK$(\tau_c)$
5:             $C \leftarrow$ GET_UPPER_CORES$(c)$
6:             $c' \leftarrow$ MAX_CORE_AFFINITY$(C, t)$
7:             MODIFY_CONFIGURATION$(\phi, t, c, c')$
8:         end if
9:     end for
10: end function
```

For each unschedulable core $c$ the task $t$ to be re-assigned is selected by function `max_score_task` (line 4). The aim is that after the re-assignment, the utilization of core $c$ is decreased as much as possible. From that perspective, it is interesting to favor tasks with a large reference utilization. On the other hand, after the re-assignment, the core $c'$ which will host the task must not increase its utilization too much. Since this core will be chosen according to its cache affinity with task $t$ (see below), $t$ is also selected based on its variability. `max_score_task` thus selects the task $t$ with the highest value of score $U(t) + V(t)$.

The set of cores $C$ candidate for the re-assignment of task $t$ are selected by function `get_upper_cores` (line 5). The cores in set $C$ are the ones hosting a critical task with a period higher than or equal to $P_t$. The core chosen $c'$ for the re-assignment is selected by function `max_core_affinity` (line 6). Core $c'$ is the one that hosts tasks with the best cache affinity with task $t$, selected as the one minimizing the following quantity:

$$|C_{t'}| - \frac{\sum_{i \in \tau_c t'} C_{i \tau_c t'}}{|\tau_{t'}|}$$

### 5.2.3 Phase 3. Finalization of cache partitioning

Up to this phase, the number of cache partitions for each core is the minimum number of partitions such that the core is schedulable. The objective of this third (optional) phase is, in case the total number of partitions is strictly less than $K$ to assign extra partitions to some cores such that their utilization is reduced. This is done iteratively as follows. First, $K$ partitions are assigned to each core. At every iteration, the core with the biggest utilization reduction is selected and its number of cache partitions is decremented unless the core become non schedulable. This process is repeated until exactly $K$ partitions are assigned.

### 6. EXPERIMENTAL RESULTS

We explore in this section the performance of PDPA and compare it with the only related algorithm, IA3 [18]. All experiments were conducted on synthetic task sets. First, the evaluation methodology is presented. Then, the determination of PDPA parameters is discussed. Finally, the performances of PDPA and IA3 are compared in different configurations (total utilization, ranges of tasks periods, etc.).

### 6.1 Methodology

#### 6.1.1 Performance metric

In order to compare the algorithms, we will use as performance metric the schedulability ratio, that is the percentage of schedulable configurations of each algorithm among the task sets generated. Every experiment will report the schedulability ratio for 1000 generated task sets.

#### 6.1.2 Task sets generation

The task sets generation process consists in generating $N$ tasks with a given total worst-case utilization of:

$$U_{no-cache} = \sum_{i=1}^{N} \frac{C_i^0}{P_i}$$

with $C_i^0$ the WCET of task $i$ with no partition reserved in the last level cache. At the end of the partitioning the total utilization is then largely lower than $U_{no-cache}$ since most, if not all, tasks will have some cache space reserved. In consequence we will test PDPA and IA3 on task sets with $U_{no-cache} > M$, with $M$ the number of cores.

Bini et al. [2] have shown that the utilization for tasks have to follow a continuous uniform law, such that schedulability tests are not biased. They also provide an efficient algorithm to generate those utilizations. This algorithm will be used to obtain the worst-case utilization of the tasks $U_i^0 = C_i^0 P_i$.

For a task $t_i$, we compute the worst WCET $C_i^0 = U_i^0 \times P_i$. With a coefficient $f$ picked randomly between 0.2 and 0.4, the best WCET is $C_i^{\ast} = f \times C_i^0$. The optimal number of partitions $\kappa_i$ is picked randomly between parameters $\kappa_{\min} \geq 1$ and $\kappa_{\max} \leq K$. Finally, the other WCETs of $t_i$ are determined: the WCETs for a number of partitions $k \geq \kappa_i$ are set to $C_i^{\ast}$. Other WCETs are generated on the line $((C_i^0, 0), (C_i^{\ast}, \kappa_i))$, with $C_i^0 = 0.98 \times C_i^0$.

### 6.2 Experimental parameters

The performance of PDPA and IA3 is evaluated using varied parameters for both the input task sets and architecture. Table 1 gives the default values for these parameters.

Additionally, PDPA uses two empirical parameters that have to be determined:
Table 1: Default values of parameters

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cores</td>
<td>M</td>
<td>4</td>
</tr>
<tr>
<td>Number of cache partitions</td>
<td>K</td>
<td>8</td>
</tr>
<tr>
<td>Size of shared last level cache (KBytes)</td>
<td>S</td>
<td>128</td>
</tr>
<tr>
<td>Minimum task period</td>
<td>$P_{min}$</td>
<td>5000</td>
</tr>
<tr>
<td>Maximum task period</td>
<td>$P_{max}$</td>
<td>150000</td>
</tr>
<tr>
<td>Minimum value for optimal number of partitions</td>
<td>$\kappa_{min}$</td>
<td>1</td>
</tr>
<tr>
<td>Maximum value for optimal number of partitions</td>
<td>$\kappa_{max}$</td>
<td>4</td>
</tr>
<tr>
<td>Number of tasks</td>
<td>N</td>
<td>20</td>
</tr>
</tbody>
</table>

- the minimal difference of periods between critical tasks: $\delta$
- the maximum number of task re-assignments before giving up returning a schedulable configuration: nbTry

We have determined the values for these parameters as follows. First, we have set nbTry to a large value ($2N$), to determine parameter $\delta$. Then, we have reduced nbTry such that a greater value makes a low difference of performance.

**Determination of $\delta$**

Parameter $\delta$ can be expressed as a fixed and a varying part:

$$\delta = \frac{P_{min} - P_{max}}{M} \times \frac{\Delta}{100}$$

with $P_{min}$ the minimal period of the input task set and $P_{max}$ the maximal period. What will be determined empirically is $\Delta$.

Figure 3 presents the schedulability ratio of PDPA for different values of $\Delta$, respectively for 10, 20 and 30 tasks in the input task set. We see that the schedulability ratio depends largely on this parameter. The schedulability ratio is maximized for almost all three configurations when $\Delta = 90$. We thus select this value for all experiments presented afterwards.

**Determination of NbTry**

With $\Delta$ set to 90%, PDPA was executed with decreasing values of parameter nbTry: $2N$, $N$, $N/2$ and $N/3$. We observed that this parameter has a very low impact on the schedulability ratio of PDPA and thus the lowest value nbTry = $N/3$ was selected.

6.3 **Comparison with IA3**

6.3.1 Schedulability ratio for different worst-case utilizations and number of cores

Figures 4, 5 and 6 present the percentage of schedulable configurations in function of the worst-case utilization of the task sets, respectively for 2 cores, 4 cores, and 6 cores. The default values of the parameters except the number of cores (see table 1) are used.

Figure 4: Schedulability ratio for different values of $U_{no-cache}$ on 2 cores

Figure 5: Schedulability ratio for different values of $U_{no-cache}$ on 4 cores

With the default parameters, PDPA always has a higher schedulability ratio than IA3, the larger difference being observed on systems with 4 and 6 cores. We attribute the higher schedulability ratio of PDPA to the fact that tasks periods are taken into account in our partitioning heuris-
Figure 6: Schedulability ratio for different values of $U_{no\text{-}cache}$ on 6 cores

tic in addition to the other task parameters that are tasks utilizations and tasks cache requirements.

6.3.2 Impact of the number of cores

For a constant total worst-case utilization of 9 and a task set made of 20 tasks, we have measured the schedulability ratio of PDPA when augmenting the number of cores in the system, all other parameters staying to their default value. Results are reported in Figure 7. We clearly see that PDPA takes benefit of more cores.

Figure 7: Schedulability ratio for different numbers of cores

6.3.3 Impact of period range

Figure 8 presents the schedulability ratio of PDPA and IA3 for different ranges of periods, with a total worst-case utilization of 6, on a system with 4 cores. The X axis is the period range (difference between the maximum period and minimum period $P_{min}$ in the task set).

As shown in figure 8, this parameter largely impacts the schedulability ratio of PDPA and IA3. With a short period range (very similar periods), IA3 outperforms PDPA. This is because in equation (3) when periods are close to each other, the dominant parameter is the task WCETs, on which IA3 performs better than PDPA because IA3 concentrates on this parameter and not on task periods. When periods are more varied, taking periods into account in the partitioning algorithm is beneficial and PDPA outperforms IA3.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented PDPA, a new algorithm for the joint task and cache partitioning problem for multicore platforms. PDPA was shown to outperform the state of the art algorithm IA3 in terms of schedulability ratio, in particular when task periods are varied. This result is obtained by taking into account, in addition to the tasks characteristics such as task utilization and cache requirements, the repartition of task periods.

One possible improvement of our work would be to consider more complex memory hierarchies, such as systems with multiple levels of shared caches, or heterogeneous cores. Another direction is to improve the representativity of task set generation and to test the algorithm on real applications.

8. REFERENCES


