CAP: Color-Aware Task Partitioning for Multicore Real-Time Applications*

Giovani Gracioli
Hardware Software Integration Lab
Federal University of Santa Catarina
Joinville, Santa Catarina, Brazil
Email: giovani@lisha.ufsc.br

Antônio Augusto Fröhlich
Hardware Software Integration Lab
Federal University of Santa Catarina
Florianópolis, Santa Catarina, Brazil
Email: guto@lisha.ufsc.br

***Abstract*—Modern multicore platforms feature multiple levels of cache memory placed between the processor and main memory to hide the latency of ordinary memory systems. The primary goal of this cache hierarchy is to improve average execution time (at the cost of predictability). The uncontrolled use of the cache hierarchy by real-time tasks may impact the estimation of their worst-case execution times (WCET). Software cache partitioning through page coloring has been considered a promising approach to isolate task workloads and thus improve WCET estimation. However, when real-time tasks share cache partitions due to false or true sharing, the inter-core delay caused by the cache coherence protocol may cause deadline losses.**

In this paper, we propose a Color-Aware task Partitioning (CAP) algorithm that assigns tasks to cores respecting their usage of cache partitions (*i.e.,* colors). Tasks that share one or more colors are grouped together and the whole group is assigned to the same processor. Thus, it is possible to avoid inter-core interference. We compared the deadline miss ratio of several generated task sets partitioned by the CAP algorithm and by the worst-fit decreasing heuristic. We executed the partitioned task sets in a modern 8-core processor with shared L3-cache using a real-time operating system. Our results indicate that a color-aware task partitioning algorithm can avoid deadline misses in a multicore processor with shared cache.

Keywords*—*task partitioning; shared cache partitioning; real-time operating systems; partitioned real-time scheduling;

I. Introduction

Current real-time embedded applications are demanding more processing power due to the evolution and integration of features that can only be satisfied by the use of a multicore processor. However, modern multicore processors have been designed to achieve maximum performance and therefore feature several latency-hiding mechanisms around shared resources. The memory hierarchy, busses, and I/O peripherals in such machines are usually bridged to the processors using caches, FIFOs, read-ahead engines, transaction builders, and other complex techniques that impact the estimation of the Worst-Case Execution Time (WCET) at the design phase (schedulability analysis) [1, 2]. Operations performed in one core may result in contention for a shared resource in other cores and unpredictably delay the execution time of tasks running on different cores. One of the main factors for unpredictability in multicore processors are shared caches (L2 and/or L3) [1]–[4].

Several recent works have been proposed to deal with shared caches and to ease the WCET estimation in multicore processors. The most common and successful approach is shared cache partitioning [2]–[6]. Cache partitioning divides the shared cache in partitions that are individually assigned to specific tasks, thus isolating the tasks' workloads that interfere with one another and leading to increased system predictability [2]–[6]. However, when cooperating tasks interact with each other by sharing partitions, the cache coherence protocol implemented in hardware invalidates shared cache lines whenever they are written and this may lead to deadline losses. In fact, a recent study carried out by the authors has shown an increase of up to 15 times on the observed WCET when tasks share cache partitions (using a modern 8-core processor with a snooping-based cache coherence protocol – see [2] for details). The consequent variations on the execution time of real-time tasks made them miss up to 97% of their deadlines [2].

Furthermore, the same study demonstrated that a partitioned real-time scheduler, in which tasks are statically assigned to cores and are not allowed to migrate to other cores, yields tasks sharing cache partitions a more deterministic behavior [2]. Partitioned scheduling for multicore real-time systems is also preferred by many researchers because each partition (or processor) is scheduled and analyzed using well-known uniprocessor scheduling algorithms and schedulability tests [7]. Moreover, partitioned approaches have better hard real-time (HRT) bounds and smaller run-time overhead than global or clustered approaches [7, 8].

In this paper, we propose a task partitioning algorithm that assigns tasks to cores according to the usage of cache memory partitions. Specifically, tasks that share one or more partitions are grouped together and the whole group is assigned to the same core, avoiding inter-core interference caused by the access to the same cache lines. We compare our partitioning strategy with an original bin packing heuristic (worst-fit decreasing) in a real processor and using a real-time operating system (RTOS). Our results indicate that our task partitioning mechanism provides HRT guarantees that are not achieved by traditional task partitioning algorithms. In summary, the main contributions of this paper are:

- We propose a Color-Aware task Partitioning (CAP) algorithm that partitions tasks to cores respecting the usage of shared cache memory partitions. Shared cache partitioning is performed by a page coloring mechanism [2, 3, 6]. We assume that each task uses a set of colors that serves as input to CAP. Then, tasks that use the same color(s) are grouped together and the entire group is assigned to the same core.
- We compare the CAP approach with an original bin packing heuristic (worst-fit decreasing) in terms of deadline misses of several generated task sets using the Partitioned-EDF scheduler. Our results indicate that it is possible to avoid inter-core interference and deadline misses by simply assigning tasks that access shared cache lines to the same core.
- We have evaluated the partitioned task sets by running

*This work was partially supported by the CAPES grant, project RH-TVD 006/2008.

them on a modern 8-core processor, with shared L3 cache using the Embedded Parallel Operating System (EPOS) RTOS. The experimental evaluation on a real machine and RTOS demonstrates the effectiveness of our task partitioning mechanism.

The rest of this paper is organized as follows. Section II provides a review on background concepts and presents the assumption and notations used in the system model. Section III briefly introduces the real-time and page coloring support provided by EPOS, which is used in the experimental evaluation. Section IV details our color-aware task partitioning. Section V shows the experimental evaluation. Section V-C discusses the main findings and Section VII concludes the paper.

II. Background and System Model

In this section we present the background needed to follow the rest of the paper and the assumptions and notations used in the proposed system model.

A. Cache Partitioning and Page Coloring

Shared cache partitioning is a method to address contention for memory spaces in (real-time) multicore systems. Cache partitioning isolates task workloads that interfere with each other, thus increasing system predictability [1]. Cache interference occurs mainly: when tasks running simultaneously on different cores evict cache lines from each other; when tasks running simultaneously on different cores access a same cache line that gets modified by one or more of them; when a task evicts cache lines from preempted ones; or when the OS unintentionally pollutes the cache while performing its duties (*e.g.,* while handling interrupts). The most common cache partitioning technique, which is used in this work, is *page coloring*. It is important to recall that interference can occur even if the main memory regions used by the tasks or by the OS are separated. For practical reasons, the cache mapping algorithms implemented in real multicore processors produce synonyms (*i.e.,* they are not fully associative), so different memory addresses can be mapped to the same cache line. This leads to a phenomenon called *false sharing*, which is a source of interference *indistinguishable* from *true sharing* from the *perspective of page coloring*.

Page coloring is a software-based technique that can be implemented on systems with a Memory Management Unit (MMU) and physically-indexed caches without any additional hardware support [1, 2, 4, 6, 9]. Page coloring explores the logical to physical address translations performed by the MMU under control of the OS so that page addresses get mapped to pre-defined cache lines. By controlling the colored bits of the cache associative set number, the OS can change the mapping of pages (usually 4 KB long) in the physical memory and the cache location. For example, the Intel i7-2600 in our platform (see Table I) has an 8 MB L3 cache that is shared among all cores. This cache uses a 16-way set-associative mapping, so each location in main memory can be stored on 16 different places in the cache. Each cache line is 64 bytes long, so there are 2^{13} sets in the cache (8 MB/16/64). Thus, the lowest 6 bits in the cache address designates a byte in a cache line, the next 13 bits designates a set, and the next 13 bits define a line from one of the 16 synonyms (or ways).

B. Task Partitioning in Partitioned Schedulers

Partitioned scheduling, such as the Partitioned-EDF (P-EDF), is usually the approach adopted for multicore real-time systems, because each core can be scheduled and analyzed using well-known uniprocessor scheduling algorithms, and because it renders better HRT bounds and lower run-time overhead than global scheduling [7, 8]. Nonetheless, the task partitioning problem is equivalent to the *bin packing* problem, which is a NP-hard problem in the strong sense [10]. Therefore, partitioning/bin packing heuristics may produce sub-optimal schedules.

The task partitioning problem can be formally defined as follows: given a set of n tasks $(T_1, T_2, \ldots, T_n)$, with different utilizations $(u_1, u_2, \ldots, u_n)$, and a finite number of processors m, each of maximum capacity of V, let each task be assigned to a processor in a way that minimizes the number of processors without exceeding the capacity V of any processor. A solution is optimal if it finds the minimal number of m[1]. The capacity of a processor depends on the scheduling policy and on the type of deadline constraint specified. For example, in an implicit-deadline task model with EDF policy, the capacity of a processor is 1 (100%), which is the schedulability bound given by the EDF schedulability test.

In principle, a task set that has total utilization smaller than the number of processors could be partitioned. However, there are cases in which a task set cannot be partitioned, even though the task set utilization does not exceed the number of processors. For example, consider the scheduling of five heavy tasks[2], with the same utilization of 0.51, on four processors. There is no partitioning algorithm able to map these five tasks into four processors. The most used bin packing algorithms for task partitioning are the *first-fit decreasing* (FFD), *best-fit decreasing* (BFD), *worst-fit decreasing* (WFD), and the *next-fit decreasing* (NFD) due to their simplicity and relative good performance [10]. In this work, we propose a task partitioning algorithm that extends these algorithms to partition tasks according to their color usage.

C. Assumptions and Notations

In this work, we consider a system with a single-chip multicore processor. The processor has m identical processors (cores) running at a fixed clock speed. M_{total} denotes the total size of the main memory available to the system. The processor has a unified last-level cache shared by all the m cores. We adopted page coloring to partition the cache at software-level —the cache is divided in N_c (number of colors) partitions. The size of each partition depends on the available memory: $S_p = \frac{M_{total}}{N_c}$.

We assume a task set τ composed of n periodic tasks. The n tasks are scheduled using the EDF scheduling policy. We also assume that all tasks share data by allocating memory from a same color (set). All data that is not shared is allocated from a unique color (set) individually assigned to each task. A task T_i is then represented as follows:

$$T_i = \left\{ e_i^{NC_i}, p_i, d_i, NC_i, M_{req}^i \right\}$$

where NC_i is the set of colors assigned to T_i, $e_i^{NC_i}$ is the WCET of T_i when it runs with $|NC_i|$ colors, p_i is the period, d_i is the relative deadline ($d_i = p_i$), and M_{req}^i is the size of the memory region allocated to T_i. Note that we assume $e_i^{NC_i}$ to encompass the run-time overhead caused by scheduling, job release, context switch, preemption, and interrupt dispatching (refer to [7, 8] for details).

The minimum number of partitions $|NC_i|$ that minimizes the WCET $e_i^{NC_i}$ depends on how partitions/colors are assigned to tasks. We assume that the values of the NC_i are known at

[1]Obviously, in real cases, the number of processor m is fixed.

[2]A task that has a utilization higher than 0.5 is considered a heavy task.

design-time. For instance, a WCET analysis tool could estimate the value of $e_i^{NC_i}$ by varying the number of colors and returning the ideal NC_i for each task T_i. It is important to mention that our focus is not to optimize the assignment of colors, but to provide a safe upper bound when tasks share colors. $e_i^{NC_i}$ is non-increasing with NC_i, which means that it begins to converge when the number of partitions reaches a point in which adding more partitions does not reduce the $e_i^{NC_i}$ [3]. The sum of the memory required for all tasks must be less than the available memory: $\sum_{i=1}^{n} M_{req}^i \leq M_{total}$.

Let N_{total} be the set of all colors used in a task set τ. $|N_{total}|$ must be less than or equal to the defined number of colors: $|N_{total}| \leq N_c$.

Let SC_i be a set of tasks that share colors with the task T_i: $\forall\ T_i, T_j, T_k \in \tau.\ (((NC_i \cap NC_j) \neq \phi) \wedge ((NC_j \cap NC_k) \neq \phi)) \rightarrow i, k, j \in SC_i$ (transitive color sharing).

We assume that the sum of the utilizations ($u_i = \frac{e_i^{NC_i}}{p_i}$) of all tasks that share a color is not greater than 100% (we call this restriction of utilizations). Formally defining:

$$\forall\ T_i \in \tau \sum_{j \in SC_i} u_j \leq 1 \qquad (1)$$

Equation 1 restricts the utilization of one or more tasks that share the same color/partition to 100%. Note that the task T_i is in the set SC_i, which implies that its utilization is also accounted. Thus, two or more tasks that share a color can be assigned to the same core, preventing them from running in parallel on different cores, and consequently, preventing the access to the same cache lines (independently of whether the sharing was true or false). Since two or more tasks will be running on the same core, they will inevitably suffer from the Cache-Related Pre-emption Delay (CRPD) caused by the lost of cache affinity after preemptions. However, the WCET $e_i^{NC_i}$ already considers the run-time overhead, including the CRPD (through a WCET estimation tool, for instance).

When tasks share a partition, the sum of the memory required by these tasks must not exceed the size of a partition, i.e., all shared data must fit into a partition. Otherwise, some data would have to be allocated from other partitions (colors), incurring in cache interference. Equation 2 presents a sufficient and necessary condition for the restriction on the shared partition size to be met. For each partition ρ, the sum of per-partition usage of tasks that share a partition does not exceed the size of one memory partition [3].

$$\sum_{\forall T_i:\ \rho \in NC_i} \frac{M_{req}^i}{|NC_i|} \leq S_p \qquad (2)$$

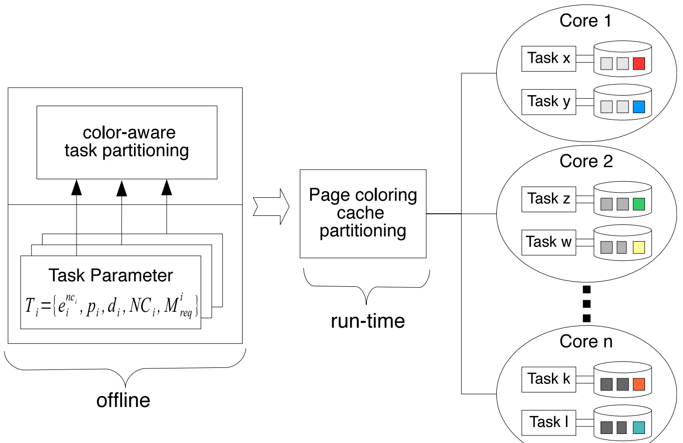
III. Real-time and Page Coloring in EPOS

We evaluated our color-aware task partitioning in a real hardware using the Embedded Parallel Operating System (EPOS) [11, 12]. EPOS is a multi-platform, object-oriented, component-based, embedded system framework implemented in C++. EPOS is the first open-source RTOS designed from scratch that supports partitioned, global, and clustered versions of EDF, RM, LLF, and DM scheduling policies [8]. A `Periodic_Thread` class represents a real-time task, aggregating mechanisms related to the periodic task re-execution. The class has a semaphore that is used by the `wait_next` method to put a thread to sleep (with an ordinary `p` operation of a semaphore) until the next period has arrived. When the timer interrupt associated with the period is triggered, the timer handler (an `Alarm`) performs a `v` operation on the semaphore to wake-up the task, thus releasing a job. A complete review of the real-time support on EPOS can be found in [8].

EPOS also supports a page coloring cache partitioning mechanism. Applications and OS memory requests are served by different heaps. When EPOS initializes, it creates applications and OS heaps that are formed by memory pages with the same color. Thus, it is possible to assign different partition to the internal OS data structures as well as to real-time tasks. Tasks allocate memory by passing a specific color to the C++ new operator: for instance, `data = new (COLOR_0) int[N]`, allocates a vector of `N` integers from the heap that has pages of color `0`. A detailed review of the EPOS page coloring mechanism can be found in [2].

IV. CAP: Color-Aware Task Partitioning

Figure 1 presents an overview of the CAP mechanism. A task set is composed of n tasks. Each task T_i has its own parameters $e_i^{NC_i}$, p_i, d_i, NC_i, and M_{req}^i. The parameters of each task serve as input to the partitioning algorithm. The partitioning algorithm finds which tasks share colors by analyzing the set NC_i of each task and uses the Equation 2 to ensure that these tasks meet the restriction of the shared partition size. The algorithm also checks whether the utilization constraints are met or not by using the Equation 1. Finally, the output of the partitioning algorithm is the assignment of those tasks that share partition(s) to the same core. Cache partitioning is performed at run-time by the EPOS page coloring mechanism described in Section III. The next paragraphs describe in details each phase of the proposed partitioning approach.



color-aware task partitioning
Task Parameter
$T_i = \{e_i^{nc_i}, p_i, d_i, NC_i, M_{req}^i\}$
offline
Page coloring cache partitioning
run-time
Core 1
Task x
Task y
Core 2
Task z
Task w
Core n
Task k
Task l

Fig. 1: Overview of the proposed color-aware task partitioning mechanism.

We describe our partitioning algorithm in a top-bottom way. The Color-Aware Partitioning (CAP) algorithm is a variation of the BFD, FFD, NFD, or WFD bin packing heuristics. We use the WFD heuristic as an example to demonstrate the algorithm. Algorithm 1 shows the pseudo code for the CAP WFD variation. The algorithm receives a task set τ, the number of available colors N_c, the total available memory M_{total}, and the number of cores m as input. The output is a boolean ($partitioned$) informing whether the task set was partitioned or not and a task set assigned to each core. The algorithm begins initializing all task sets as empty and setting $partitioned$ to true (lines 1 and 2). Then, the task set τ is partitioned into groups by calling the function $FindTasksWithSharedColors$, which returns the groups of tasks that share colors in the array $taskGroups$ (line 3). The algorithm tests if each group of tasks satisfies the

Equation 1 and 2 and returns false if a group does not satisfy them (lines 4 to 9). After guaranteeing both restrictions, the groups are sorted by decreasing order of utilizations[3] (line 10) and partitioned using the WFD heuristic (line 11). Finally, it creates and returns a task set for each partition by using the assigned groups. Note that it is possible to use any partitioning heuristic in line 11. The difference is that we are partitioning groups of tasks and not individual tasks.

Algorithm 1 CAP_WorstFitDecreasing(τ, N_c, M_{total}, m)

Input: τ: a task set of "n" tasks as described in Section II-C, N_c: available number of colors/partitions, M_{total}: available memory in the system, m: number of cores in the processor
Output: $partitioned$: a boolean if the WFD heuristic is able to partition the task set, $task\_set[m]$: a task set per core

```
1: $task\_set \leftarrow \emptyset$                    ▷ Initializes all task sets as empty
2: $partitioned \leftarrow true$
3: $taskGroups \leftarrow$ FindTasksWithSharedColors($\tau$)
4: for each group of tasks in $taskGroups$ do
5:     if group does not satisfy Eq. 1 and Eq. 2 then
6:         $partitioned \leftarrow false$
7:         return $\{partitioned, task\_set\}$
8:     end if
9: end for
10: Sort groups by decreasing order of utilization
11: Apply the WFD heuristic by groups
12: $task\_set \leftarrow$ Create task set per core
13: return $\{partitioned, task\_set\}$
```

Algorithm 2 shows the pseudo code for the function $FindTasksWithSharedColors$. It receives a task set as input and returns a multi map data structure ($taskGroups$) representing the groups of tasks that share colors[4]. For example, $taskGroups[0]$ contains all tasks that share at least one color in the group 0, $taskGroups[1]$ contains all tasks that share at least one color in the group 1, and so on. Tasks in different groups do not share any color. The function first initializes all array positions with zero (line 1). Then, it compares the set of colors for each task and fills a specific array position whenever two tasks use at least one common color (lines 2 to 12). For instance, if $A_\tau[0][1]$ is equal to one, it means that task T_1 shares color with task T_0. If it is zero, T_0 and T_1 do not share color. The function ends calling another function $MapTasksToGroups$ to create the groups of tasks and returns the groups (lines 13 and 14).

Algorithm 2 FindTasksWithSharedColors(τ)

Input: τ: a task set of "n" tasks
Output: $taskGroups$: a multi map data structure representing groups of tasks that share colors

```
1: $A_\tau[n][n] \leftarrow [0,0]$ ▷ An array representing the colors usage pattern among tasks. Initializes all positions with 0
2: for $i \leftarrow 0$ to $n$ do
3:     for $j \leftarrow 0$ to $n$ do
4:         if $i \neq j$ then
5:             $has\_shared\_data \leftarrow \tau[i].NC_i \cap \tau[j].NC_j$   ▷ $NC$ is the set of colors
6:             if $has\_shared\_data == true$ then
7:                 $A_\tau[i][j] \leftarrow 1$
8:                 $A_\tau[j][i] \leftarrow 1$
9:             end if
10:        end if
11:    end for
12: end for
13: $taskGroups \leftarrow$ MapTasksToGroups($\tau$, $A_\tau$)
14: return $taskGroups$
```

The function $MapTasksToGroups$ is depicted in Algorithm 3. It receives a task set and an array representing the colors usage pattern among tasks as input. The output is a multi map data structure $taskGroups$ representing groups of tasks that share colors. For each task T_i in τ, the function verifies if the current task T_i is in a group and if it is not, a new group is created and T_i is inserted in this group (lines 4 to 7). Then, for each task T_j in τ, if T_i and T_j share a color ($A_\tau[i][j] = 1$) the function calls $SolveSharedColorsChain$, passing the multi map $taskGroups$, the current T_j index (j), the task set τ, the current $group$, and the array A_τ (lines 8 to 14). $SolveSharedColorsChain$ adds all tasks that share color(s) with T_j recursively. Note that if T_i does not share colors with another task, it is assigned to a new group without any other task. Finally, $taskGroups$ is returned to $FindTasksWithSharedColors$, containing tasks mapped to groups and respecting the colors usage pattern among them.

Algorithm 3 MapTasksToGroups(τ, A_τ)

Input: τ: a task set of "n" tasks, A_τ: an array representing the colors usage pattern among tasks
Output: $taskGroups$: a multi map data structure representing groups of tasks that share colors

```
1: $taskGroups \leftarrow \emptyset$                 ▷ Initializes all groups as empty
2: $group \leftarrow -1$
3: for each task $T_i$ in $\tau$ do
4:     if $T_i$ is not in $taskGroups$ then
5:         $group = group + 1$                ▷ Creates a new group of tasks
6:         map $T_i$ into $taskGroup[group]$
7:     end if
8:     for each task $T_j$ in $\tau$ do
9:         if $A_\tau[i][j] == 1$ then              ▷ $T_i$ shares data with $T_j$
10:            if $T_j$ is not in $taskGroups$ then
11:                SolveSharedColorsChain($taskGroups, j, \tau,$
12:                $group, A_\tau$)   ▷ Inserts all tasks that share colors into the same group
13:            end if
14:        end if
15:    end for
16: end for
17: return $taskGroups$
```

The last function, $SolveSharedColorsChain$, is described in the pseudo code of Algorithm 4. It receives a multi map data structure with the task groups, the index of the current task being analyzed in the $MapTasksToGroups$ function, a task set τ, the current $group$ id, and the array A_τ with the colors usage pattern among all tasks in the task set τ. There is no output. The objective of $SolveSharedColorsChain$ is to add all tasks that share at least one color into the same group recursively. For each task T_i in τ, if T_i shares a color with T_{index} ($A_\tau[index][i] = 1$), the function performs two tests. First, it tests if T_{index} is already in the current group, and inserts T_{index} into $taskGroups[group]$ when T_{index} is not in the group. This test is necessary because T_{index} may have been inserted before (lines 3 to 5). The second test verifies if T_i is in the $taskGroups[group]$. When the test fails, it means that T_i has not been analyzed yet. Then, T_i is added to $taskGroups[group]$ and $SolveSharedColorsChain$ is called again, with i as the new $index$. Thus, all tasks that share a color with T_i are added to the same group. The process is repeated until finishing to add all tasks that share at least one color to the same task group.

V. Experimental Evaluation

This section describes the experimental evaluation of the CAP algorithm. The objective is to compare its performance in terms of HRT guarantees with that for traditional bin packing partitioning algorithms. All experiments were performed on an Intel i7-2600 processor (see Table I) on top of EPOS RTOS.

A. Experiment Description

We randomly generated task sets similar to those in [2, 4], adding a color number for each task. We selected the

[3]A group utilization is the sum of the utilizations of all tasks in that group.

[4]It is important to keep in mind that, although the *placement new operator* in C++ is used at run-time to dynamically allocate memory, its specialization to implement the page coloring mechanism in EPOS relies on constant color aliases that are known at compile-time.

Algorithm 4 SolveSharedColorsChain($taskGroups$, $index$, τ, $group$, A_τ)

Input: τ: a task set of "n" tasks, $index$: index of the task being analyzed, $group$: current group with tasks that share colors with T_{index}, A_τ: an array representing the colors usage pattern among tasks

```
1:  for each task $T_i$ in $\tau$ do
2:      if $A_\tau[index][i] == 1$ then
3:          if $T_{index}$ is not in $taskGroups$ then
4:              map $T_{index}$ into $taskGroup[group]$
5:          end if
6:          if $T_i$ is not in $taskGroups$ then
7:              map $T_i$ into $taskGroup[group]$
8:              SolveSharedColorsChain($taskGroups$, $i$, $\tau$,
9:              $group$, $A_\tau$)
10:         end if
11:     end if
12: end for
```

TABLE I: Intel i7-2600 processor features.

Clock speed	3.4 Ghz
Cores	4
SMT (hyperthreading)	2 per core (8 logical cores)
L1 cache	4 x 64 KB 8-way set-associative (32 KB separate data and instructions caches)
L2 cache (non-inclusive)	4 x 256 KB 8-way set-associative (unified)
L3 cache (inclusive)	8 MB 16-way set-associative (unified)

periods uniformly from {25, 50, 75, 100, 150, 200} ms and utilizations uniformly from [0.1, 0.7]. The WCET of a task was defined according to the generated period and utilization. As our processor has 8 cores, we fixed the number of task groups and colors to 8 (one color for each task group). For each task, we randomly selected a color between [1, 8]. In order to respect the utilization constraints in the CAP algorithm, we generated tasks of the same color until the accumulated utilization reached the [0.9, 1.0] interval. We used the P-EDF scheduling algorithm with implicit-deadlines. We used the CAP WFD variation, because with the WFD heuristic each core tends to have similar utilization. Note that the generated task sets have total utilization close to HRT bounds. The objective was to run the same task sets using both partitioning approaches and then analyze whether deadlines were met or not.

Each task allocates an array to represent its *working set* and subsequently performs random read and write operation on it in a loop. The working set size (WSS) was adjusted varying the size of the array. Figure 2 shows part of the corresponding source code. Each task iterates for 200 times (ITERATIONS variable). We considered scenarios with four different WSS (ARRAY_SIZE variable): 32 KB, 64 KB, 128 KB, and 256 KB. We defined a write ratio of 20% (one write for each four readings) and 33% (one write for each two readings). Increasing the write ratio, we stimulate the cache coherence protocol to invalidate shared cache lines more often.

The number of reads and writes of a task varies depending on the WCET. To adjust the number of repetitions of each task, we varied the "repetitions" parameter according with [2]. We also adjusted the repetitions parameter according with the WSS to account for the intra-task (self-evictions) and intra-core (preemption delay) cache interferences. When a task evicts its own cache lines, it increases its execution time and eventually misses a deadline as shown in [2]. Each task allocates memory for the array using a color received as parameter.

B. Percentage of Missed Deadlines

To evaluate the ratio of missed deadlines for the CAP WFD and the original WFD algorithms, we generated 10 task sets. The number of tasks in a task set varies from 23 to 30 tasks. The number of tasks in a task group varies from 2 to 6. We then partitioned each task set using the CAP WFD and WFD

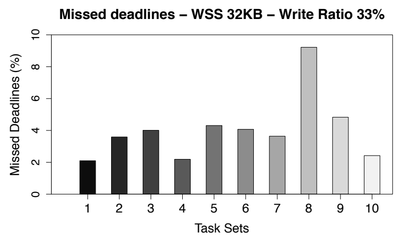
```
#define MEMORY_ACCESS 16384
#define WRITE_RATIO 2 // or 4
int job(unsigned int repetitions , int id, int color) {
  int sum = 0, *array;
  Pseudo_Random * rand;
  rand = new (color) Pseudo_Random();
  array = new (color) int[ARRAY_SIZE];
  for(int i = 0; i < ITERATIONS; i++) {
      Periodic_Thread :: wait_next();
      for(int j = 0; j < repetitions ; j++) {
          for(int k = 0; k < MEMORY_ACCESS; k++) {
              int pos = rand->random() % (ARRAY_SIZE - 1);
              sum += array[pos];
              if ((i % WRITE_RATIO) == 0) array[pos] = k + j;
          }
      }
  }
}
```

Fig. 2: Part of the task function source code.

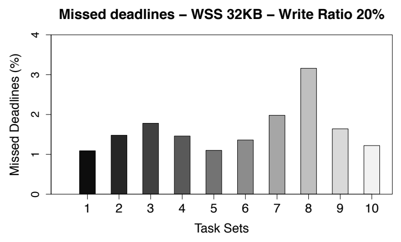
algorithms. We executed the 10 different partitioned task sets 50 times, varying the WSS and write ratio as previously described. Then, we extracted the number of missed deadlines for each partitioned task set from those executions (*over 88 hours of tests using a real hardware and RTOS*). In EPOS, the `Alarm` is responsible for releasing a task by calling a `v` operation on an associated `Semaphore` (see Section III). Since we assume implicit deadlines, we counted a missed deadline whenever the `v` operation was issued for the semaphore while its value was greater or equal to zero. In practice this means that a new task's job was released before the previous job had finished.

Figure 3 shows the percentage of missed deadlines. Figure 3(a) shows the results for a WSS of 32 KB for the write ratio of 33% and Figure 3(b) for the write ratio of 20%. On the x-axis, we vary the task set. On the y-axis, we present the percentage of missed deadlines for each task set. All Figures show missed deadlines only for the WFD heuristic, because *all tasks in all task sets partitioned by the CAP WFD algorithm were able to meet their deadlines*. Note that Figure 3(a) and 3(b) have different scales for the y-axis, because decreasing the write ratio causes the cache coherence protocol to invalidate less cache lines, improving the system performance. However, for both write ratios, all task sets partitioned by WFD missed deadlines: from 2.10% (task set 1) to 9.22% (task set 8) for write ratio of 33% and from 1.09% (task set 1) to 3.16% (task set 8) for write ratio of 20%. In general, the task sets have similar performance for WSS of 32 KB. The task set 8, in special, have two task groups with five tasks in each group. When the task set is partitioned by the WFD algorithm, the tasks in each group are assigned to different cores, causing contention on the arrays and increasing their execution times. The CAP WFD, in contrast, avoids the contention and meets the application HRT constraints.

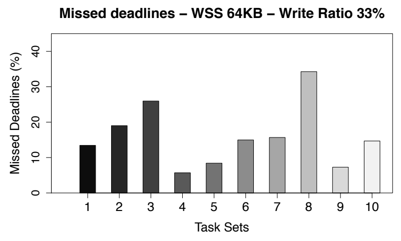
Figure 3(c) shows the percentage of missed deadlines for a WSS of 64 KB and write ratio of 33% and Figure 3(d) for WSS of 64 KB and write ratio of 20%. For write ratio of 33%, the deadline miss ratio varies from 5.70% to 34.27% and for write ratio of 20% it varies from 4.53% to 29.42%, both for the task sets 4 and 8, respectively. Comparing this behavior with the behavior observed for WSS of 32 KB, the percentage of missed deadlines increases due to the array size. Our processor has four 256 KB, 8-way L2-caches and one 8 MB, 16-way shared L3-cache with 64 bytes per cache line and 8192 sets. With a WSS of 32 KB, each task demands eight pages of the same color. With a WSS of 64 KB, each task demands 16 pages of the same color. Consequently, there is more contention for the cache ways both in L2- and L3-cache. The CAP WFD is able to decrease this contention by preventing tasks that share a color from running in parallel on different cores at the same



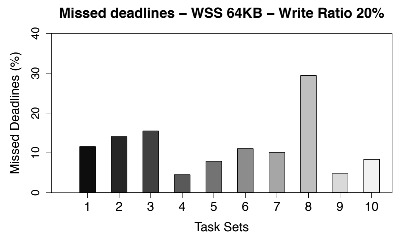
(a)



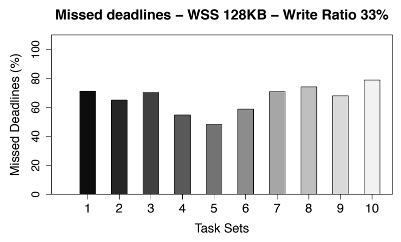
(b)



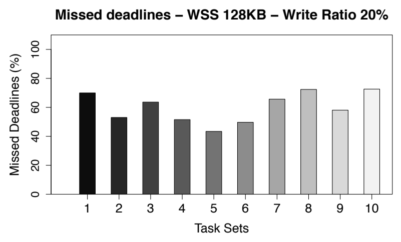
(c)



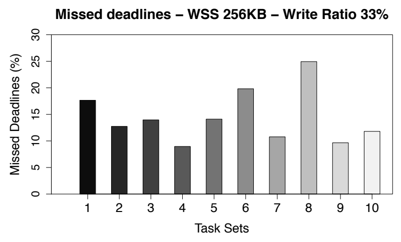
(d)



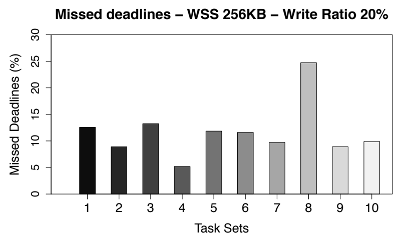
(e)



(f)



(g)



(h)

Fig. 3: Percentage of missed deadlines of the WFD heuristic, varying the WSS and write ratio.

time. Thus, the cache coherence protocol does not delay the tasks by invalidating cache lines due to false sharing. The only delay a task may experience is the preemption delay (lost of cache affinity after a preemption) and intra-task cache misses in the L2-cache (caused by the task's own data accesses).

Figure 3(e) shows the percentage of missed deadlines for a WSS of 128 KB and write ratio of 33% and Figure 3(f) for write ratio of 20%. In Figure 3(e), the deadline miss ratio varies from 48.17% (task set 5) to 78.84% (task set 10) and in Figure 3(f) it varies from 43.39% (task set 5) to 72.60% (task set 10). Comparing this behavior with the two previous experiments, the percentage of missed deadlines increased even more. This is due to the same reasons as for the increase from 32 to 64 KB. With a WSS of 128 KB, each task demands 32 pages from the same color, causing more cache contention and more preemption delay due to cache lines that were evicted after preemptions. It is worth mentioning again that we adjusted the repetitions parameter (see Figure 2) as we increased the array size in order to decrease the application execution time and account for the intra-task cache misses. Moreover, with 128 KB, and two tasks running on different hyperthreads, the L2-cache was more often full and more L2-cache misses occurred.

Finally, Figure 3(g) shows the percentage of missed deadlines for a WSS of 256 KB and write ratio of 33% and Figure 3(h) for write ratio of 20%. For write ratio of 33%, the deadline miss ratio varies from 8.96% (task set 4) to 24.94% (task set 8) and for write ratio of 20% it varies from 5.19% (task set 4) to 24.73% (task set 8). In comparison with the previous results, the percentage of missed deadline has decreased. With WSS of 256 KB, the preemption delay is greater, which means that we had to reduce the number of repetitions of each task. Thus, there is less contention for the cache lines within the same color. However, the deadline miss ratio for a WSS of 128 KB is similar to that of 64 KB.

C. Discussion

Below we discuss our main observations on the experiments:

- **Color-Aware task Partitioning:** our CAP algorithm assigns tasks to processors respecting the usage of colors by tasks. We generated 10 different task sets that were partitioned using the CAP WFD and the original WFD algorithms. We ran each partitioned task set for 50 times varying the WSS of each task from 32 to 256 KB.

From these executions, we extracted the percentage of missed deadlines. By simply assigning tasks to cores, CAP prevented tasks from accessing shared cache lines and thus met the HRT application requirements without missing any deadline.

- **Working set size:** we analyzed the behavior of the chosen multicore platform by varying the WSS (32, 64, 128, and 256 KB). With 128 KB, the tasks have the greatest deadline miss ratio due to the more contention for shared cache lines and preemption delay. With 256 KB, the preemption delay is higher due to L2-cache misses. For 32 KB, mostly of the tasks data fits into the L2-cache and the deadline miss ratio is smaller than in the other WSSs.
- **Write ratio:** we ran the partitioned task sets using two different write ratios (20 and 33%). By changing the write ratio, we stimulate the cache coherence protocol to invalidate more or less cache lines [2]. For all WSS variation, the execution with the write ratio of 20% has missed less deadlines as expected. Moreover, the change of memory-bus direction for read and write operations also has a negative impact on performance and isolation of tasks on different cores. For write ratio of 33%, the change in bus direction is more often than in write ratio of 20%, which increases the execution time.
- **True and false sharing:** the application used in our experiments does not explicitly share data. Tasks allocate memory using a color received as argument. Some tasks allocate memory from the same color, as described in SectionV-A, causing concurrent accesses to shared cache lines (false sharing). In an application with shared data, a mutual exclusion protocol, such as the *flexible multiprocessor locking* protocol [13], must be used to ensure correctness. In such applications, the blocking times caused by priority inversion and the implicit delay caused by the cache coherence protocol must also be taken into account while estimating the WCET. Such protocols can be easily accommodated under EPOS page coloring mechanism, which provides a straightforward API to allocate specific memory partitions to shared data.
- **Colors assignment:** in our task model (see Section II-C), we assume that tasks have a set of colors that represents memory partitions. The process of assigning colors to tasks is a complex optimization problem. To assign the optimal combination of colors, one must analyze the data usage for each task and thus select the best combination of colors that minimizes the cache delays for a specific processor. However, once an heuristic is chosen, EPOS can easily incorporate it under its memory management abstractions, encapsulating the specificities of the underlying architecture in well-defined microcomponents. Memory partitioning is not restricted to applications. It can also be used for internal OS data structures, thus enabling a fine-grain color assignment and providing OS isolation.
- **Other scheduling policies:** we ran our experiment using the P-EDF scheduling policy. It is also possible to apply CAP to any other scheduling policy, such as RM and DM. The difference would be restricted to the processor capacity given by each scheduling algorithm.
- **Other partitioning heuristics:** we executed our experiments using the WFD heuristic. However, as stated before, it is possible to apply any partitioning heuristic in CAP algorithm. The strategy would be forming groups of tasks that share colors and then applying a partitioning heuristic using the entire group instead of single tasks.
- **Run-time overhead:** since task partitioning by CAP is performed offline, it does not add any overhead to the associated scheduler run-time overhead. Previous works have shown that a partitioned scheduler (P-EDF) has a smaller run-time overhead than clustered (C-EDF) and global (G-EDF) approaches, which improves the schedulability ratio of HRT applications [7, 8].
- **Utilization restriction:** CAP limits the utilization of a task group to 100%. This restriction may reduce the applicability of the algorithm. However, the restriction is necessary to provide HRT guarantees that were not achieved before. *CAP is a first attempt to reduce the cache coherence effects on real-time tasks and to provide HRT guarantees* (experimentally proved).
- **WCET estimation and synthetic workloads:** we estimated the WCET of the synthetic task sets through experiments. To the best of our knowledge, no existing static analysis technique is able to account for the effects of the coherence protocol. Also, there is no estimation tool for the processor used in our experiments. These are the reasons why we estimated the WCET experimentally. Furthermore, it is a common practice in multicore real-time research works to use synthetic workloads due to the lack of benchmarks and applications with real-time constraints.

VI. Related Work

The Largest Working set size First, Grouping (LWFG) is a partitioning algorithm for SRT systems that evenly distributes the working sets of tasks on all cores to reduce cache conflicts and capacity-related cache misses [14]. LWFG uses NFD heuristic, ordering the tasks by their decreasing WSS. If the NFD heuristic fails to find a core with sufficient capacity to partition the group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. Our CAP algorithm assumes that page coloring is available and each task uses a set of colors. Thus, it is possible to determine which tasks share cache lines by inspecting the colors of each task. In contrast, in the LWFG approach, it is not clear how the algorithm is aware of data sharing among tasks. Moreover, LWFG is proposed for SRT systems, since partition is retried with a smaller group whenever NFD fails. Thus, inter-core interference is not avoided, which may incur in deadline losses. CAP limits the utilization of a task group to 100% to avoid inter-core interference. Furthermore, incorporating the idea of task group splitting in CAP is straightforward, and thus provide similar performance for SRT systems as LWFG. CAP is designed to be integrated to any bin packing heuristic, including the NFD. The performance evaluation of LWFG was carried out using a real-time patch for the Linux kernel, which may limit the observed gains due to the inherent non real-time behavior of Linux. Instead, we evaluated CAP using an RTOS, without the excessive run-time overhead introduced by Linux.

Nemati *et al.* also proposed a partitioning algorithm that groups tasks with shared resources and assigns the entire group to a same core [15]. Resource sharing among tasks is identified by a matrix. By analyzing this matrix, the algorithm calculates a task weight through a cost function. Then, tasks are ordered according to their weight and inserted into a task group if it satisfies a schedulability test. Although similar to our algorithm, the proposed approach was not evaluated, neither simulated nor in a real hardware.

Suzuki *et al.* proposed two algorithms, one based on solving a mixed-integer linear problem and another one based on solving a variant of the knapsack problem, for allocating tasks to processor to decrease conflicts at the cache and DRAM bank levels [16]. The proposed algorithms use cache and bank coloring together. The objective is to optimize the assignment of cache colors to tasks and bank colors to processors and avoid cache and bank interference among tasks. The authors have shown by experimentation that cache and bank coloring together increases the system performance. However, the allocation algorithms were not evaluated in a real hardware nor in terms of real-time guarantees. Moreover, transitive color sharing seems not to be treated by the proposed memory interference model. Our page coloring mechanism could be easily extended to also support bank coloring.

Software cache partitioning for uniprocessor real-time systems was first proposed by Wolfe [5]. Code and data of a task are logically restricted to memory portions that map into cache lines assigned to the task (similar to page coloring). Liedtke *et al.* used page coloring to also provide predictability for uniprocessor real-time systems [6]. Bui *et al.* considered the cache partitioning problem as an optimization problem whose objective is to minimize the worst-case system utilization under the constraint that the sum of all cache partitions (based on page coloring) cannot exceed the total cache size [17]. A genetic algorithm solves the optimization problem. Compiler-based techniques partition the shared cache by accessing the source code and allocating partitions to tasks at compile time [18].

Kim *et al.* proposed a cache management scheme in which each core has a set of private partitions to avoid inter-core cache interference [3]. However, tasks within each core can share cache partitions. Each task can have a larger number of partitions which could improve its execution performance. Additionally, memory partitions can be utilized by all tasks. They bound the penalties due to the sharing of cache partitions by accounting for them as CRPDs when performing the schedulability analysis. This technique can result in intra-core interference (*i.e.,* self-evictions).

MC^2 treats the management of cache lines as synchronization and scheduling problems [4]. The proposed approach uses page coloring and real-time multiprocessor locking protocols together. The OS associates a set of colors to each task. With locking mechanism, a task locks its set of colors whenever it is invoked. Under the scheduling mechanism, cache colors are treated as preemptive resources in which concurrent accesses are mediated by scheduling. The authors compared both techniques in terms of schedulability of the P-RM algorithm and concluded that cache locking approach is better than the scheduling approach. In our previous work, we extended the page coloring performance analysis to P-EDF, C-EDF, and G-EDF algorithms and also considered the impact of the OS on the WCET [8]. Mancuso *et al.* proposed a memory framework that uses profiling techniques to analyze the memory access pattern of tasks and obtain the most frequently accessed memory pages [1]. Then, cache locking and page coloring are used to isolate tasks and increase predictability. This work also used Linux and requires hardware with cache locking support.

Calandrino *et al.* proposed a cache-aware scheduling algorithm that provides SRT guarantees [19]. At run-time, hardware performance counters (HPCs) estimate the WSS of tasks. Then, tasks are scheduled in a way that cache thrashing is avoided. Two assumptions limit the usage of the proposed approach: applications must not consume more memory than the shared cache size and tasks do not share data. FP_{CA} is another cache-aware scheduling algorithm that divides the shared cache space into partitions [9]. Tasks are scheduled in a way that at any time, any two running tasks' cache spaces are non-overlapped. A task can execute only if it gets an idle core and enough cache partitions. However, both proposed cache-aware real-time schedulers use a task model that does not allow the sharing of memory partitions by tasks.

VII. Conclusion

Shared cache memory partitioning is an efficient approach to increase predictability of multicore real-time systems. However, when real-time tasks share memory partitions, the delay caused by the cache coherence protocol may result in deadline losses. In this work, we have proposed a task partitioning algorithm that assigns tasks to cores respecting their usage of cache partitions. Tasks that use at least one common partition are grouped together and the whole group is assigned to an individual processor, avoiding inter-core interference. We have compared the deadline miss ratio of generated task sets partitioned by our algorithm and by the WFD heuristic. The ratio was obtained running the partitioned task sets in a real processor with an RTOS. The results have indicated that it is possible to avoid deadline misses by simply assigning tasks that share cache partitions to the same processor. As future work, we plan to investigate how *hardware performance counters* can provide run-time information to improve real-time guarantees by avoiding inter-core interference. The source codes of CAP and EPOS are available online [12].

References

[1] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Proc. of the RTAS '13*, 2013, pp. 45–54.

[2] G. Gracioli and A. A. Fröhlich, "An experimental evaluation of the cache partitioning impact on multicore real-time schedulers," in *Proc. of the RTCSA '13*. IEEE, 2013, pp. 441–450.

[3] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Proc. of the ECRTS 2013*, 2013, pp. 80–89.

[4] C. Kenna, J. Herman, B. Ward, and J. H. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS '13*, 2013, pp. 157–167.

[5] A. Wolfe, "Software-based cache partitioning for real-time applications," *J. of Comp. Sofw. Engi.*, vol. 2, no. 3, pp. 315–327, Mar 1994.

[6] J. Liedtke, H. Haertig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *RTAS '97*, 1997, pp. 213–223.

[7] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *Proc. of the RTSS'10*. IEEE, 2010, pp. 14–24.

[8] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, pp. 669–714, 2013.

[9] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. of the EMSOFT*, 2009, pp. 245–254.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

[11] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[12] EPOS. (2014, Jun) Epos. [Online]. Available: http://epos.lisha.ufsc.br

[13] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA '07*, 2007, pp. 47–56.

[14] C. Lindsay, "LWFG: A cache-aware multi-core real-time scheduling algorithm," Master's thesis, Virginia Poly. Inst. and State Uni., 2012.

[15] F. Nemati, M. Behnam, and T. Nolte, "Efficiently migrating real-time systems to multi-cores," in *ETFA '09*, Sept 2009, pp. 1–8.

[16] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *CSE '13*. IEEE, 2013, pp. 685–692.

[17] B. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *Proc. of the RTCSA '08*, 2008, pp. 101–110.

[18] X. Vera, B. Lisper, and J. Xue, "Data caches in multitasking hard real-time systems," in *Proc. of the RTSS'03*. IEEE, 2003, pp. 154–165.

[19] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *Proc. of the ECRTS '09*. IEEE, 2009, pp. 194–204.