Inter-Task Cache Interference Aware Partitioned Real-Time Scheduling

Zhishan Guo
University of Central Florida
zsguo@ucf.edu

Kecheng Yang
Texas State University
yangk@txstate.edu

Fan Yao Amro Awad
University of Central Florida
{Fan.Yao,Amro.Awad}@ucf.edu

ABSTRACT

With the increasing number of cores in processors, shared resources like caches are interfering task execution behaviours more heavily and often render global scheduling approaches infeasible in practice. While partitioned scheduling alleviates such interference, in most existing partitioned approaches, constant WCET, which potentially includes all possible interference, must be statically pre-determined prior to the partitioning processes. In this paper, we show that by taking inter-task interference into consideration when making scheduling decisions, resource efficiency can be significantly improved in both temporal and spatial domains for multi/many-core real-time systems. In particular, we propose the inter-task interference matrix (ITIM) to model the inter-task cache/memory interference in a pair-wise manner. Focusing on the problem of interference-aware partitioned scheduling with ITIM, we formalize it as a mixed integer linear program (MILP), which can be solved to achieve optimal solution at the cost of high computational complexity. Meanwhile, we also provide several polynomial-time algorithms to solve the problem approximately. We extensively profile a set of WCET benchmark programs on x86-based multiprocessor server to collect ITIM. The algorithms are evaluated comprehensively, and the evaluation results demonstrate the superior performance of the proposed approaches under various settings.

1 INTRODUCTION

With the growing gap between the speeds of memory and processing cores, incorporating more cores within processor chips becomes a dominant trend. In fact, single-core processors are predicted to be obsolete in a few years. Multi-core systems allow hiding the memory latency through parallel threads and many applications running concurrently. Moreover, multi-core processors allow efficient data sharing through shared last-level caches (LLCs) and memory modules. Multi-core processors have been designed with area and power efficiency in mind, and thus many hardware resources are shared between cores within the processor chip. While such sharing allows more efficient use of caches for shared memory multi-threaded applications, it also introduces new sources of contentions that can impact the timing of real-time tasks. As real-time systems typically require worst-case guarantees in their temporal behaviours, the growing interference with huge variations must be modelled and taken care of in system verification. In fact, accounting for such contentions and preventing them from causing inter-task interference becomes a strict requirement when multi-core processors are employed in avionic systems and safety-critical applications, where strict certification and testing requirements are mandated.

In real-time scheduler design, there are two fundamental approaches: global scheduling and partitioned scheduling. Global scheduling algorithms have a simple single ready-queue implementation and solid, classic theoretical schedulability results [6] [8]. However, the run-time behaviours of tasks may be unpredictable due to the flexibility of migrations. Both the observed run-time overheads [7] and the provisioned worst-case execution time (WCET) [28] may be unacceptably large under global scheduling, and this effect becomes more severe as the number of cores and the number of tasks grow. In addition, the $1/m$-based analyses [6] [8] for global scheduling algorithms have recently been shown to be ineffective, even theoretically compared to a very simple partitioned scheduling heuristic [10]. On the other hand, under partitioned scheduling (e.g., Partitioned Earliest-Deadline-First (P-EDF) [14] and Partitioned Rate Monotonic (P-RM) [25]), tasks are statically assigned to processors and a particular uniprocessor scheduler (e.g., uniprocessor EDF or RM) is applied on each individual processor. Therefore task behaviours under partitioned scheduling become simpler than that under global scheduling. A natural question arises: does partitioned scheduling solve the aforementioned issues about interference on multiprocessor platforms? Unfortunately, the answer is complicated and in general "no".

With partitioned scheduling, tasks can be delayed due to contention on shared resources that are shared by cores. Such contention makes it hard to predict task execution times [30]. In order to eliminate inter-core interference, the state-of-art techniques for partitioned scheduling typically implement a certain level of isolation between processors and/or tasks for shared resources. Particularly, shared (last level) caches are one of the most commonly shared hardware resources by cores, and are partitioned via cache hardware features (e.g., set partitioning [32] and way allocation [15]). Many existing works partitions the last-level cache so that *each task* has a dedicated portion of the shared cache [18] [4] [23]. While these techniques reduce the inter-core cache interference into (almost) the minimum level, they limit the share of each task and cause under-utilization of cache; cache partitioning typically assumes the maximum cache occupancy of a specific application, which leads to certain inefficiency and undertuilization of cache. Other alternative approaches allow certain tasks (e.g. with same criticality level [24]) to share a larger portion of the last-level cache. This, on

SAC '20, March 30-April 3, 2020, Brno, Czech Republic
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3374014

the contrary, allows two tasks to execute simultaneously on different processors, where the interference on the last-level shared cache can be substantial—leading to a much larger worst-case utilisation of the system.

Besides inter-core cache interference, *inter-task* cache interference (e.g., cache related preemption delays) can also have a non-trivial impact on real-time tasks. Such cache interference (e.g., in private caches) exist for tasks that are scheduled on the same processor. In fact, several prior studies like [13] have shown that such interference may constitute up to 33% of the task's total execution time. To mitigate the inter-task interference issues, one potential approach is to again partition private caches (e.g., L1 cache) in each core for tasks scheduled on the same core. For instance, Altmeyer *et al.* [4] studies the tradeoff between task's execution time and size of cache allocations, and presents a cache partitioning algorithm that is optimal with respect to task set schedulability. While last-level cache partitioning is generally viable due to its excessive capacity, private caches isolation can severely exacerbate tasks' execution times when their accessible private caches drop below certain threshold [22]. This is especially the case for processors in real-time systems where private caches are often designed with significantly smaller sizes to enable fast access.

In short, isolating every single task for each hardware resource by itself is neither efficient nor necessary. In other words, *as far as a certain degree of isolation is maintained between the tasks that can heavily interfere with each other, the overall system performance could stay in a well-balanced manner.*

Based on this observation, we take a somewhat intermediate approach: instead of partitioning shared caches among all tasks, the last level cache is partitioned to *each processor* such that tasks assigned to the same processor have access (in turns) to the well-sized portion of the shared cache *associated with the processor.* Specifically, we leverage the way allocation technique (i.e., Intel CAT [16]) to partition last-level cache for individual cores to eliminate inter-core cache interference. Under such a partitioned scheme, the goal is to come up with a technique that ensures tasks with strongly interfering access to dedicated per-core resources (including private cache and the share of last-level cache) are not grouped to the same core. We propose an inter-task interference-aware scheduling framework on multiprocessors, where tasks with high cache interference when running on the same core are judiciously mapped onto distinct cores. The algorithm finally performs inter-task interference aware scheduling to maintain the performance of real-time tasks while keeping the system better utilised under a partitioned manner.

Our main contributions are summarised as follows.

- To represent the inter-task cache interference and guide the task-to-core partition, we propose the inter-task interference matrix (ITIM), which characterises the inter-task cache interference and system interrupt delay between each pair of tasks. ITIM provides a safe upper bound to inter-task interference.
- We formulate the inter-task interference-aware schedulability problem into a mixed integer linear program (MILP). We then transform the problem into a graph cut one and propose a swapping based approach with a resource capacity bound of $1 - \frac{1}{m}$ (where m is the number of processors). Based on this idea, we further enlarge the search space in each round and form a Genetic Algorithm based approach.
- We measure the inter-task interference using the Malardalen WCET benchmark [20] on a testbed equipped with Intel Xeon v4 processors, and conducted interference aware schedulability tests under various parameter settings for the proposed approaches.

Organisation. The rest of this paper is structured as follows: Section 2 presents our system and interference model. Section 3 formalize the interference-aware partitioned scheduling problem as a mixed integer linear program (MILP). Section 4 transforms this problem to a k-CUT graph problem variant and proposes an algorithm to solve it approximately. Section 5 improves the approximate solution by introducing a genetic algorithm based approach. Section 6 presents our experiment results to evaluate the proposed approaches. Section 7 concludes this work.

2 SYSTEM MODEL AND ASSUMPTIONS

Workload model. We consider a predefined workload which can be characterised by a set of n tasks $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$. Each task τ_i may release an infinite number of jobs, each of which has the same worst-case execution time (WCET) requirement C_i (known as the WCET of the task), and consecutive releases must be at least T_i time units apart (known as the *period* of the task). In this work, we focus on implicit-deadline tasks only. That is, each job of task τ_i has an *absolute* deadline at T_i time units after its release, or equivalently, each task τ_i has a *relative deadline* of T_i time units.

Note that in real-time systems analysis, it is typical that overhead and interference are measured and included in the WCET—that is *not* the case here. In this paper we use C_i to denote τ_i's WCET when running in isolation (in a cold cache); i.e., *excluding* cache-related interference and system overhead, and we call it the ***plain* WCET** of τ_i. Under a certain partition, the WCET of τ_i may be inflated to be $\hat{C}_i \geq C_i$ that safely takes cache-related interference and system overhead into account, and we call $\hat{C}_i$ the ***effective* WCET** of τ_i. Similarly, we define the *plain* utilisation of τ_i by $u_i = C_i/T_i$ and define the *effective* utilisation of τ_i by $\hat{u}_i = \hat{C}_i/T_i$.

Without loss of generality, we index tasks by their periods (therefore, also their relative deadlines) in a non-decreasing order, i.e., $\forall i \leq j, T_i \leq T_j$. The following lemma shows that a lower-indexed task could possibly preempt a higher-indexed task under the above task index scheme, no matter whether P-EDF or P-RM scheduling is applied.

Lemma 2.1. *Under either P-EDF or P-RM scheduling, if task τ_i preempts task τ_j, then it must be true that $i < j$, provided that tasks are indexed such that $\forall i \leq j, T_i \leq T_j$.*

Proof. This lemma is trivially true for P-RM scheduling. As each task is assigned a fixed priority, it is clear that only higher-priority task could possibly preempt low-priority ones.

For P-EDF, let us consider some job J_i of task τ_i that is released at time r_i and has an absolute deadline at d_i preempts some job J_j of task τ_j that is released at time r_j and has an absolute deadline at d_j. Because under P-EDF scheduling, jobs' priorities are determined by absolute deadlines, it must be true that $d_i \leq d_j$ in order to allow J_i to preempt J_j. Also, J_i must be released after J_j is released, i.e.,

$r_i > r_j$; otherwise, J_i would have prevent J_j from execution from J_j is released until J_i is complete and therefore it is impossible for J_i to *preempt* J_j. Combining $d_i \leq d_j$, $r_i > r_j$, and the fact that tasks have implicit deadlines, we have $T_i = d_i - r_i < d_j - r_j = T_j$, which implies $i < j$ under our task index scheme. □

System Model. We consider a system that consists of m homogeneous processors, which are the typical setting from mainstream chip vendors. Each processor has its own small but fast private caches, and all processors share one large but slower last-level cache. We note that simultaneous multi-threading (SMT) is disabled, and tasks do not compete for on-core logic components (such as ALU). Each processor runs a single task at one time. Furthermore, the total number of tasks out-numbers the number of cores; i.e., $n > m$. The system uses a fully preemptive partitioned scheduling algorithm and the task set is assumed to have been loaded into the memory at the beginning of run time.

Memory Systems. Caches are one of the most commonly shared hardware in multiprocessors. Typically, cache architectures fall into two categories: *direct-mapped cache* and *set-associative cache.* As a first attempt, this paper focuses on **direct-mapped cache**, where each cache contains multiple sets and each set has exactly one cache line. Upon a cache miss, the data in the memory will be loaded and stored into the corresponding cache set. When a task is preempted more than once by other tasks, the accumulative interference needs to be determined. Note that for direct-mapped cache, the cache block evictions are deterministic as each cache set only has a single line. Therefore, the overall interference could be effectively bounded by the summation of their pair-wise interference values. While in a set-associative cache, where each cache set has multiple cache lines called *ways*, this may no longer hold. This is because modern processors typically come with complex cache-line replacement policies. When there is a cache miss, the corresponding data block can be placed in any of the ways (based on eviction policy) in the corresponding set. Under such policies, the overall interference may well exceed the mutual impact between any of the two tasks involved in the pre-emptions. We leave this as future work. We assume that off-chip memory is large enough to hold the whole workload. To ensure the same mappings to cache occur on each run, we assume huge pages (2MB) and the disable of Address Space Layout Randomization (ASLR). Note that since using huge pages is mandatory to minimize the overheads of virtual memory translations, and given its wide support in almost all commercial processors, such an assumption is practical. Disabling ASLR is a common practice when reproducible behavior is anticipated, and hence it needs to be disabled for real-time systems to minimize non-determinism.

Remark 2.1. *Most contemporary multiprocessors share the last level cache across processors. Different levels of restrictions regarding inter-core cache interference can be put on how applicable the assumptions are. In our work, the last-level cache is partitioned among cores, and thus no inter-core cache interference exists. By default, each core gets an equal share through way partitioning, while further optimisations can be performed for cache-to-core mapping, this is out of the scope of this paper—please refer to e.g., [11] [29], for work focusing on optimising the cache partition to reduce the inter-core interference. Orthogonal to the existing work, our work mainly focuses on finding the better task-to-cache/core mapping (task partitioning) in terms of inter-task interference and partitioned schedulability.*

Model and Problem: Inter-Task Interference Matrix. As mentioned in Section I, due to the resource sharing under a multi-core setting, the inter-task interference cannot be simply ignored. The competition between the tasks of the shared resource could hugely affect the overall system performance. When conducting the partitioning, if we can isolate most pairs of tasks with strong interference, the extra overhead caused by resource competition can be reduced significantly, leading to much better overall system performance and resource efficiency.

To quantify the inter-task interference, we propose Inter-Task Interference Matrix (ITIM) $M \in \mathbb{R}_+^{n \times n}$, where n is the number of tasks. Each element in the matrix $M_{i,j}$ ($i < j$), also known as the *interference utilisation*, can be defined as:

$$M_{i,j} = \frac{\hat{C_j}^{(i)} - C_j}{T_j} \tag{1}$$

where $\hat{C_j}^{(i)}$ is τ_j's inflated worst-case execution time (WCET) including all overheads when τ_j and τ_i run together, while C_j is τ_j's plain WCET when running alone (in a cold cache). Accroding to Lemma 2.1, $M_{i,j} = 0$ when $i \geq j$. Note that the inter-task interference will vary under different system states regarding the memory and cache. The memory address of storing the tasks' data, which directly maps to a specified cache block, can strongly affect the inter-task interference. In the experiments section, we applied this measurement based approach to derive the interference utilisation by running several WCET benchmark programs on a server platform (See Section 6).

To better justify the model, here we provide an alternative method to derive the interference utilisation via analysing evict cache blocks (ECB) and useful cache blocks (UCB) statically for each task. Here we briefly introduce the method while for more details, please refer to [26] and [3].

Given a predefined task set, we can calculate the interference score using the following formula:

$$M_{i,j} = \left\lceil \frac{T_j}{T_i} \right\rceil \times \max_k |\mathrm{UCB}_j \cap \mathrm{ECB}_i^{(k)}| \times \frac{\gamma}{T_j} + \epsilon \tag{2}$$

where $\max_k |\mathrm{UCB}_j \cap \mathrm{ECB}_i^{(k)}|$ is the maximum cardinality of the UCB and ECB's intersection for all (k) programming points in higher priority task τ_i, γ (in time unites) is the worst additional interference delay per cache block, and ϵ is the additional time cost (system interrupt) per preemption. Equation (2) means that $M_{i,j}$ depends on the number of preemptions and the cache blocks that task j needs to reload after task i finishes in the worst-case.

Example 2.1. *Given a predefined task set which is shown in Table 1, assuming $\gamma = 0.15$, we apply Equation (2) for all pair of tasks:*

$$M_{1,2} = \left\lceil \frac{3}{2} \right\rceil \times |\{2,3\} \cap \{2,3,4\}| \times \frac{0.15}{3} = 0.2$$

$$M_{1,3} = \left\lceil \frac{6}{2} \right\rceil \times |\{2,3\} \cap \{1,2,3,4\}| \times \frac{0.15}{6} = 0.2$$

$$M_{2,3} = \left\lceil \frac{6}{3} \right\rceil \times |\{2\} \cap \{1,2,3,4\}| \times \frac{0.15}{6} = 0.05$$

Table 1: Parameters of a task set.

Task	C_i	T_i	*ECB*	*UCB*
τ_1	*1*	*2*	*{2,3}, {2}*	*{1,2,3}*
τ_2	*1*	*3*	*{2}, ∅*	*{2,3,4}*
τ_3	*1*	*6*	*∅*	*{1,2,3,4}*

$$M = \begin{bmatrix} 0 & 0.2 & 0.2 \\ 0 & 0 & 0.05 \\ 0 & 0 & 0 \end{bmatrix}.$$

Problem. For a multi-core real-time system, given the task set τ and the interference matrix M, our goal is to find a system partition that involves task partitioning, such that real-time correctness, can be guaranteed on each core under the consideration of interference overheads.

Mathematically, given the total number of processors m and a task set τ of n tasks with the inter-task cache interference matrix $M_{n \times n}$, our goal is to obtain a feasible partition $\mathcal{P}$ (an n to m mapping) of the task set where $\mathcal{P}(i) \in \{1, 2, ..., m\}$ means task τ_i is assigned to processor $\mathcal{P}(i)$, i.e.,

$$\forall p = 1, 2, ..., m : \sum_{i|\mathcal{P}(i)=p} \frac{C_i}{T_i} + \sum_{i,j|\mathcal{P}(i)=\mathcal{P}(j)=p} M_{i,j} \leq \beta; \quad (3)$$

where β is the utilisation threshold for the corresponding uniprocessor scheduling algorithm: $\beta = 1$ if the system is scheduled under earliest deadline-first (EDF), while $\beta = n_{max}(2^{\frac{1}{n_{max}}} - 1)$ when the system is scheduled rate monotonically (RM), where n_{max} is the maximum number of tasks assigned to one processor under a certain partition.

In the following sections, we provide three potential solutions for this problem.

3 A MIXED INTEGER LINEAR PROGRAMMING FORMULATION

Given the ITIM for a certain system, we are able to form up the cache-aware partitioned scheduling problem as a mixed integer linear programming (MILP) problem as follows.

We first introduce $n \times m$ *integer* variables $\{x_{i,p}\}$ for $1 \leq i \leq n$ and $1 \leq p \leq m$ that are subject to the following two sets of linear constraints.

Constraints Set (i):

$$\text{For } i = 1, 2, \ldots, n \text{ and } p = 1, 2, \ldots, m, \quad x_{i,p} \geq 0.$$

Constraints Set (ii):

$$\text{For } i = 1, 2, \ldots, n, \quad \sum_{p=1}^{m} x_{i,p} = 1.$$

By Constraint Sets (i) and (ii), $\{x_{i,p}\}$ becomes a set of binary variables (i.e., each of them is either 0 or 1), and each $x_{i,p}$ intuitively indicates whether task τ_i is assigned to processor p. Constraint Set (ii) also indicates that each task must be assigned to and only to a single processor.

We then introduce $n \times n \times m$ auxiliary real-number variables $y_{i,j,p}$, which are subject to the following constraint sets.

Constraints Set (iii):

$$\forall 1 \leq i < j \leq n, 1 \leq p \leq m, y_{i,j,p} \geq 0.$$

Constraints Set (iv):

$$\forall 1 \leq i < j \leq n, 1 \leq p \leq m, y_{i,j,p} \geq x_{i,p} + x_{j,p} - 1.$$

Constraints Set (v):

$$\forall 1 \leq j \leq i \leq n, 1 \leq p \leq m, y_{i,j,p} = 0.$$

Given the binary variable $\{x_{i,p}\}$ indicating the task-to-processor assignment under partitioned scheduling, Constraint Sets (iii), (iv), and (v), in fact, make $y_{i,j,p} \times M_{i,j}$ an upper bound on the interference on processor p caused by task τ_i preempting task τ_j.

When $i < j$, Constraint Sets (iii) and (iv) indicates that if task τ_i and τ_j are both assigned to processor p, then $y_{i,j,p}$ is at least 1; otherwise, $y_{i,j,p}$ is at least 0. When $i \geq j$, Constraint Set (v) simply assigns constant 0 to $y_{i,j,p}$, because as shown in Lemma 2.1, only a lower-indexed task could possibly preempt a higher-indexed task under the task indexing scheme presented in Section 2.

We finally introduce one last auxiliary *real-number* variable z that captures the maximum *effective* utilisation on a single processor and serves as the objective function to be minimised.

Constraints Set (vi):

$$\text{For } p = 1, 2, \ldots, m, \quad \sum_{i=1}^{n} \left(x_{i,p} \cdot u_i \right) + \sum_{i=1}^{n} \sum_{j=1}^{n} \left(y_{i,j,p} \cdot M_{i,j} \right) \leq z.$$

The above Constraint Set (vi) is built upon the fact that given the upper bound of $y_{i,j,p} \times M_{i,j}$ on the interference on processor p caused by task τ_i preempting task τ_j, the total effective utilisation processor p can be upper bounded by

$$\sum_{i=1}^{n} \left(x_{i,p} \cdot u_i \right) + \sum_{i=1}^{n} \sum_{j=1}^{n} \left(y_{i,j,p} \cdot M_{i,j} \right).$$

Thus, the MILP can be formed up as:

$$\begin{aligned} &\textit{Minimise} \quad && z \\ &\textit{Subject to} \quad && \textbf{Constraint Sets (i) - (vi)} \end{aligned}$$

As a uniprocessor scheduling algorithm with utilisation bound β is used on each processor, the system is partitioned schedulable if and only if $z \leq \beta$. In addition, if the MILP finds the optimal value of z is at most β, the corresponding values of $\{x_{i,p}\}$ directly indicate such a task-to-processor partition under which the system is schedulable.

In this MILP, $\{u_i\}$ and $\{M_{i,j}\}$ are constants. There are nm integer variables — $\{x_{i,p}\}$ and $n^2m + 1$ real-number variables — $\{y_{i,j,p}\}$ and z. Also, Constraints Set (i) has n inequality linear constraints, Constraints Set (ii) has n equality linear constraints, each of Constraints Sets (iii) and (iv) has $\frac{(n-1)n}{2} \times m$ inequality linear constraints, Constraints Set (v) is just constant-value assignments for variables, and Constraints Sets (vi) has m inequality linear constraints. In total, there are n equality linear constraints and $n + (n-1)nm + m = O(n^2m)$ inequality linear constraints.

4 GRAPH K-CUT BASED PARTITIONING

Due to the NP-hardness of the problem [13] (as well as its MILP representation), our next goal becomes to find algorithms that can identify near-optimal task partitions in polynomial or pseudo-polynomial time. This subsection transforms the task partition problem into a graph cutting problem and describes an insightful swapping-based search heuristic. We first introduce the original graph max-k-cut problem, then describe an existing approximation algorithm, then describe how the problem transformation is done and give an example to demonstrate it, and finally describes our solution.

Max k-CUT and an approximation solution. Let $G(V,E)$ denote a graph which has $|V| = n$ vertices, and k be a positive integer less than n. Each vertex v_i contains a value $v_i.val$ and edge $e = (u,v) \in E$ connect two vertices $u, v \in V$ with weight $w(u,v)$. A k-CUT is a partition to the vertices into k subsets and can be represented as a n-to-k mapping $\mathcal{P}$, where $\mathcal{P}(i)$ indicates the processor ID that task τ_i is allocated to. The objective is to minimise the total weight of the edges in each subset; i.e.,

$$\min_{\mathcal{P}} \Big(\sum_{u,v|\mathcal{P}(u)=\mathcal{P}(v)} w(u,v) \Big); \tag{4}$$

For this problem, Random Swap [17] has the best known approximation bound in terms of total weight. The idea is quite simple, we start with an arbitrary k-CUT (i.e., partition into k sets), randomly choose a pair of vertices that belongs to two different partitions, and attempt to make a swap—if the swap could lead to a better partition in terms of overall remaining weights, then the swap is confirmed. We go into the next round and choose another pair of vertices randomly, until no swap could lead to any improvement of the overall objective function.

The following lemma proved by Gaur *et al.* in [17] indicates that this algorithm has a nice approximation ratio in terms of total cut-off weights.

Lemma 4.1 (Capacity Bound). *The graph partition algorithm could obtain a remaining total weight no smaller than* $1 - 1/k$ *(k is the number of cuts) of the optimal solution when no other constraints exist, i.e.,*

$$\frac{W_{\mathcal{P}'}}{W_{\mathcal{P}^*}} \geq 1 - \frac{1}{k}, \tag{5}$$

where $W_{\mathcal{P}'}$ *is the total weight obtained by the graph partition algorithm,* $W_{\mathcal{P}^*}$ *is the optimal total weight.*

Problem transformation. The transformation works in the following manner: each task τ_i is treated as a vertex v_i in a directed graph, with its utilisation $u_i = C_i/T_i$ as the weight of the vertex, while the interference utilisation from τ_i to τ_j ($M_{i,j}$) represents the weight of the directed edge between i and j

The interference minimisation problem (described in Subsection III-A) becomes to partition ('cut') the vertices in G into subsets, so that the sum of weights of remaining edges (that are not part of any 'cut') is minimised (while the cut off weights are maximised). Meanwhile, note that our problem put additional constraints that the partition has to be feasible, i.e., the total value (utilisation) of the vertices (tasks) in a subset must not exceed a certain scheduler-specific threshold (1 for EDF and ln2 for RM).

Example 4.1. *Consider the task set shown in the following table, with a given TIM:*

$$M = \begin{bmatrix} 0 & 0.07 & 0.09 & 0.041 \\ 0 & 0 & 0.04 & 0.02 \\ 0 & 0 & 0 & 0.08 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Table 2: Parameters of a task set.

Task	*WCET (C_i)*	*Period (T_i)*	*utilisation (u_i)*
τ_1	*1*	*2*	*0.50*
τ_2	*1*	*3*	*0.33*
τ_3	*2*	*4*	*0.50*
τ_4	*5*	*10*	*0.50*

Figure 1 illustrates the directed graph after transformation, as well as one possible partition.

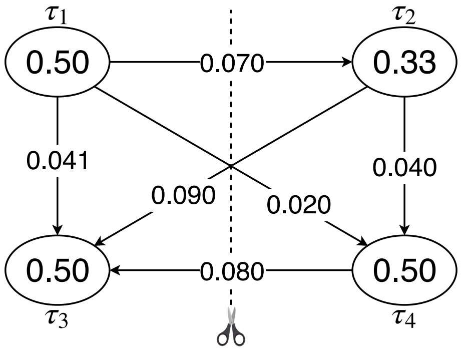

Figure 1: Directed graph as a result of transformation for the task set shown in Table 2. The direction of edge (τ_i, τ_j) indicates the priority level order $\tau_i \triangleright \tau_j$, and the weight of the edge denotes the interference utilisation. The dash line in the middle of the figure shows one possible task partition: {1,2,2,1}

Max k-CUT for interference minimisation. Other than schedulability constraints, it makes sense to try to minimise the total interference. Inspired by the existing approach for max k-CUT problem, we propose a similar swapping-based algorithm with pseudo-code shown in Algorithm 1.

Note that the 'if' and 'elseif' conditions inside the while loop of this algorithm guarantees that once a feasible partition is reached via swapping, further swapping will only be accepted if the result partition remains feasible.

5 GENETIC ALGORITHM BASED PARTITIONING

The approximation solution of the graph K-Cut problem, as described in the previous subsection, is essentially a random walk in a feasible solution space. Unfortunately, due to the nature of the problem that there exists massive amount of local minima, we expect

Algorithm 1 Max k-CUT for Interference Minimisation

```
function GRAPH_PARTITION(M, $\tau$, n)
    Arbitrarily form a k-CUT (partition) $\mathcal{P}$ to $\tau$
    flag = True
    If_succeed = False
    while flag do
        flag = False
        for any pair of tasks $\tau_i, \tau_j, \mathcal{P}(i) \neq \mathcal{P}(j)$ do
            if max percore utilisation can be reduced then
                swap $i$ and $j$ in $\mathcal{P}$
                flag = True
            elseif overall interference can be reduced under same max percore utilisation then
                swap $i$ and $j$ in $\mathcal{P}$
                flag = True
            end if
        end for
    end while
    if If_succeed == True then
        return $\mathcal{P}$
    else
        return -1
    end if
end function
```

it to perform poorly in many scenarios (verified in experiments). Enlarging the candidate partitions considered in each round during a random walk would help avoid local minima from a statistical point of view. Inspired by this fact, this subsection presents how Genetic Algorithm can be adapted to handle the problem of interest.

Genetic Algorithm is a heuristic search mechanism to solve optimisation and search problems with multiple local optima. It is inspired by the process of natural selection (in biology). The typical stages include selection, crossover, and mutation and are introduces below for our problem.

Chromosome (partition). A chromosome is a string $\{a_1, a_2, ..., a_n\}$ that defines a potential partition, where $a_i \in [1, m]$ indicates the processor ID that task τ_i is assigned to. For instance, there are three tasks and two cores, where τ_0 is in Core 1, τ_1 and τ_2 are in Core 2, the chromosome will be {1,2,2}.

Fitness Function. The fitness function provides a measure to the quality of all chromosomes. For real-time correctness guarantees, typically the objective is binary: 1 for schedulable and 0 otherwise. Here we overcome such limit by evaluating a partition by the maximum per-core utilisation, including interference overhead; i.e.,

$$f(\mathcal{P}) = \max_{1 \le p \le m} \left(\sum_{i<j, \mathcal{P}(i)=\mathcal{P}(j)=p} M_{i,j} \right). \tag{6}$$

Similar to the MILP described in Subsection III-B, if the fitness function of the finalised chromosome (partition) is no greater than 1 (or $ln2$, respectively), the system becomes feasible under P-EDF (P-RM, respectively).

Initialisation. At the beginning, POP_SIZE (large integer, default $n(n+1)/2$) number of candidate partitions are generated randomly, by arbitrarily assigning a core to each task of each candidate.

Selection. In each round, among all candidates, according to a predetermined retention rate parameter ($\in (0, 1)$, default 0.5), we keep a portion of the chromosomes with higher fitness scores and discard the rest. Among the ones that are kept, we will choose enough[1] pairs of chromosomes (called parents) according to proportionate roulette wheel selection [5]: the ones with higher fitness scores are more likely to be selected. Specifically, the probability of selecting partition $\mathcal{P}_i$ is set as:

$$\begin{aligned}\text{Prob}(i) &= \frac{S - f(\mathcal{P}_i)}{\sum_{j=1}^{N_R} (S - f(\mathcal{P}_j))} \\ &= \frac{S - f(\mathcal{P}_i)}{(N_R - 1)S},\end{aligned} \tag{7}$$

where N_R is the size of the population that is kept in each round (typically half the total population size) and $S = \sum_{k=1}^{N_R} f(\mathcal{P}_k)$ is the total population fitness score.

Crossover. For each selected pair of parent chromosomes, we generate a new pair of child chromosomes according to the following two steps: (i) Choose a random task ID from 1 to n; then (ii) exchange the assignment for all tasks with a larger ID. For example, the crossover of {1,3,1,2,3,1} and {1,1,2,3,1,2} at ID 4 results in {1,3,1,2,1,2} and {1,1,2,3,3,1}.

Mutation. We give each child a (small) chance to mutate itself according to MUTATION_RATE ($\in (0, 1)$, default 0.05). For each task assignment of each child chromosome, if (with a small probability) it is chosen to receive a mutation, we randomly assign it a processor (overwrite the original assignment). This step gives the whole system extra chance to escape local minima. For example, a chromosome {1,3,1,2,3,1} may mutate to {1,3,1,2,2,1} or {1,3,1,2,1,1} (or remain the same) when task τ_5 is chosen for a mutation.

Iteration. For the new population (kept parent chromosomes as well as newly generated child chromosomes), we iterate the whole process from the Selection step, until the number of generations reached to a predefined upper bound N_GENERATION (default $n\lg n$).

The pseudo code for genetic algorithm is shown at Algorithm 2.

6 EVALUATION

In this section, we demonstrate the experiment setup and evaluation. Specifically, Subsection 6.1 shows the experimental setup, including system configuration, workload, and how the inter-task interference matrix is measured. It also discusses the baseline approaches that we aim to compare against in this work. While Subsection 6.2 reports performances of the proposed approaches under various conditions.

6.1 Experimental Setup

System configuration. To profile inter-task interference, we setup a x86-based server that is equipped with Intel Xeon E5-2686 v4 processors. The processor has 18 per-core *256KB 8-way set-associative* L1/L2 caches and one *45MB 20-way set-associative* last level cache[2]. To avoid contention on logic components in each physical core,

[1] Here enough means that an exact number of child chromosomes can be generated, so that the total population maintains at POP_SIZE after the round.

[2] We employ this many-core system to mimic future embedded processors that may potentially have computation power comparable to server class processors.

Algorithm 2 Genetic Algorithm for Task Partition

```
function GENETIC-ALG($M$, $\tau$, POP_SIZE, N_GENERATION, N_R, MUTATION_RATE)
    Generate POP_SIZE partitions (POP) randomly
    for gen from 1 to N_GENERATION do
        pop = N_R better partitions in POP
        while pop size < POP_SIZE do
            Select p1, p2 from pop according to (7)
            pos = generate a random integer from 1 to n
            c1 = p1[0: pos] + p2[pos+1:n]
            c2 = p2[0: pos] + p1[pos+1:n]
            for each pos in c1 and c2 do
                if rand() < MUTATION_RATE then
                    c1 (or c2)[pos] = random int (1 to n)
                end if
            end for
            pop = pop ∪ c1 ∪ c2
        end while
        POP = pop
    end for
end function
```

Simultaneous Multi-threading has been disabled to ensure that only one task can run on each physical core at a time. We leverage Intel CAT userspace interfaces (i.e., *pqos*) [21] to partition the last level cache by allocating 2 ways to each of the 8 cores, which are the subset of the available cores configured to dedicatedly run real-time tasks. Note that CAT has been recently adopted in real-time multi-core scheduling components such as Xen VMM [2]. To enable real-time scheduling policy, we set up a linux v4.15 based OS employed with the *deadline task scheduling* that implements the EDF algorithm [1]. Real-time tasks are configured with the `SCHED_DEADLINE` policy. To eliminate performance variations due to hardware runtime power management mechanisms (e.g., P states and C states), we have disabled core sleep states, turned off Turbo Boost and configured the frequency governor to use the highest frequency all the time [12].

Task set configuration and ITIM generation. We utilised part of the well known Malardalen WCET benchmark [20] as our tasks. We discarded 26 of the 38 programs as they are too ‘mini’, i.e., the execution cost is smaller than the scheduling granularity supported in the current system (i.e., 1024 nanoseconds resolution) of the implementation, and thus the periodicity of the task cannot be precisely controlled. We directly used 6 programs from the benchmark–`adpcm, bsort100, edn, lms, matmult, ndes`, while the following ones are modified by adding a outside loop to scale their execution times by a factor of 4-10: `compress, expint, fft1, ns`. Other parameters of the task set are measured/generated as follows:

- The number of cores $m = 4$. The number of tasks $n = 10$ unless otherwise specified. Note that the time complexity of the MILP solver is quite high ($\sim O(m^n)$), such that for massive amount of Monte Carlo tests, one cannot afford to have a large n or m. Moreover, the current setting, which is by no means comprehensive, already shows clear trends for approach comparison purposes.
- The plain WCET of each task is measured by executing each selected program 100 times and take the observed maximum duration of execution length.). Note that we have intentionally omitted the first measurement of the execution time, which is considerably larger than the subsequent measurements due to additional OS-level activities (e.g., page faults and buffer cache misses).
- The inflated WCET of a task under each other task is measured by executing the combined pair of programs 100 times and taking the observed maximum duration of overall execution length (including overhead).
- The utilisations (and thus periods) of the tasks are generated by the UUnifast algorithm [9]. Note that for multiprocessor, the utilisation generated by the UUnifast algorithm cannot guarantee each task’s utilisation is less than β. We always discard a naturally infeasible task set and regenerate.

Each element of the interference matrix is calculated according to (1). To profile the pair-wise interference, we select two real-time tasks each time, and control one task to preempt the other task at a random time instant for 100 times. Note that tasks are indexed by their (randomly generated) periods. Also, we did not restrict the priority order of the benchmark tasks—they are determined by the randomly generated periods according to Lemma II.1. We used utilisation based RM or EDF schedulability test [27] for each uniprocessor.

Approaches implemented.

Mixed-ILP: The mixed-integer linear programming described in Sec. III-B.

Greedy: By worst-fit heuristic, in each round we try to allocate the unassigned task with largest utilisation onto the processor with smallest possible ID, while making sure that the targeted processor remains schedulable.

K-Cut: The graph K-cut approach described in Sec. III-C.

Genetic-Alg: The genetic algorithm based approach described in Sec. III-D, under default parameter settings.

6.2 Experimental Comparison Study

Benchmark workload. We randomly assign periods (and thus utlisations) to the benchmark workload, such that for each system overall plain utilisation level, we generated 1000 different task sets. Given a task set and interference utilisation, different approaches may return different (task-to-core) partitions, some are valid while some are not. For such evaluation, the higher this ratio gets, the better the approach performs.

Figure 2 reports the percentage of each of the 1000 sets (under certain system plain utilisation) that are P-EDF schedulable (with inter-task interference) under each of the proposed approaches. Among four approaches, the mixed-integer linear programming method outperforms the rest (as we have shown its optimality in Theorem III.1) and greedy approach performs the last (due to simple heuristic applied).

We are glad to notice that the graph k-cut based approach outperforms genetic algorithm. This is likely due to the fact that graph

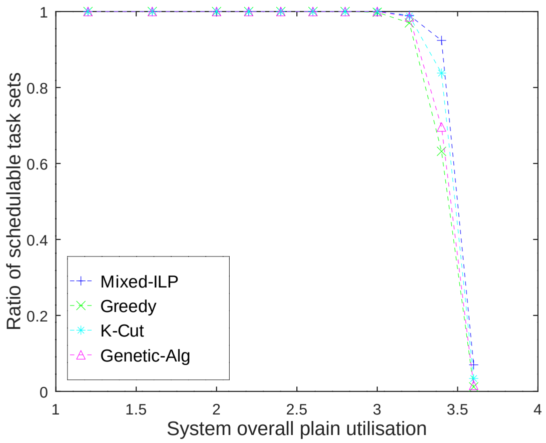

Figure 2: Schedulability ratio comparison of under P-EDF with varying system utilisations.

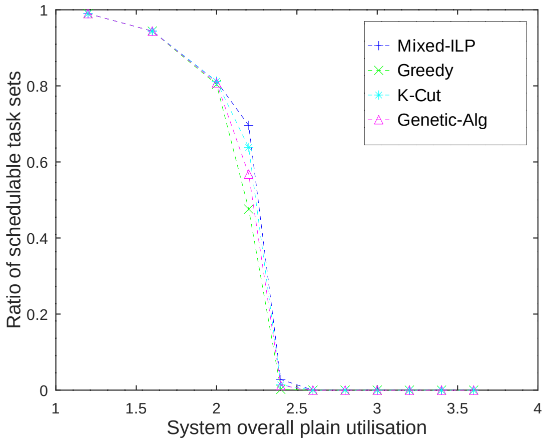

Figure 3: Schedulability ratio comparison under P-RM with varying system utilisations.

k-cut does not stop its search whenever a swap can lead to improvement in terms of maximum per-core utilisation. Note that genetic algorithm is a randomised search and its performance varies from time to time and depend upon parameter settings. Since we are conducting extensive evaluation over massive amount of task sets, we cannot afford to have a large population and a larger number of generations for the approach—these hugely limits the performance of Genetic Algorithm. We also would like to point out that there are task sets that are schedulable by Genetic Algorithm but not K-Cut (and the other way around).

Figure 3 reports the performance of the same sets under P-RM, where the approaches remain at the same order comparing to that of P-EDF. A major difference is that since we are using utilisation based schedulability test for uniprocessor RM, the schedulability ratio drops much sooner at around 60% overall utilisation (interference cause approximately 10% extra utilisation).

Benchmark workload with inflated interference. Different system settings may result in various interference levels between tasks. In order to test how the proposed approaches work for more general scenarios other than the benchmark workload, we artificially

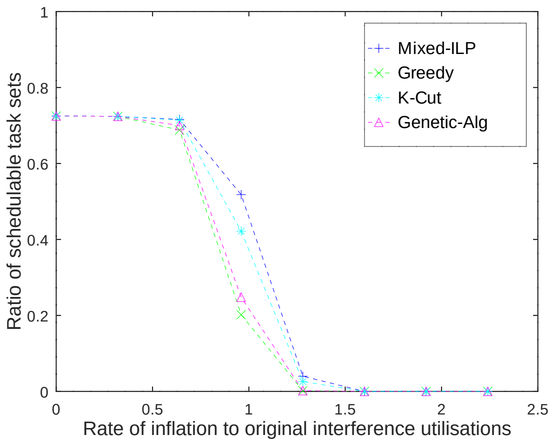

Figure 4: Schedulability ratio comparison under P-RM with varying interference utilisations.

adjusted the interference level by multiplying the matrix by a factor ranging from 0 to 2.25, with system overall plain utilisation at around the threshold values in terms of schedulability ratio.

Figure 5 reports the performance of the approaches under P-RM, where the x axis represents the factor for modifying ITIM. It is expected that when interference utilisation becomes larger, schedulability ratio drops for all approaches. We also notice that the relative performance relationship remains stable under this setting. This figure also demonstrated that when system is closer to be fully utilised (e.g., here plain overall utilisation is set to 2.2), then more than approximately 5% to 10% extra interference utilisation may cause significant performance drop. Similar behaviours are identified in P-EDF, while the performance drawings are omitted due to space limit.

Large-scale synthetic task sets. We understand that the benchmark provides very limited size of task set. In order to validate the performance of the approaches under a broader setting, we include synthetic workload and interference, enlarge the number of processors to 8, and set plain system utilisation to 2.2. Note that MILP method runs exponentially slow and thus are omitted for this setting when $n > 16$.

Figure 5 reports the performance of the approaches under P-RM, where the x axis represents the number of tasks per set. We notice that when task number is closer to $mlogm$, the performance tends to be better. This is because (i) when task number is small, to have a plain system utilisation of 2.2, we easily ends up with $m + 1$ or more large tasks that cannot pack in m cores, while (ii) when task number is too large, the number of pairs of tasks per core increases quadratically, such that interference utilisation dominates the processor, making it unfeasible.

7 CONCLUSION

In this paper, we focused on partitioned scheduling that takes inter-task interference into consideration while making partition and scheduling decisions. We proposed the concept of ITIM to represent inter-task interference in a pair-wise manner. We presented an MILP formulation to exactly solve this problem of interference-aware partitioned scheduling. Due to the potentially high time complexity

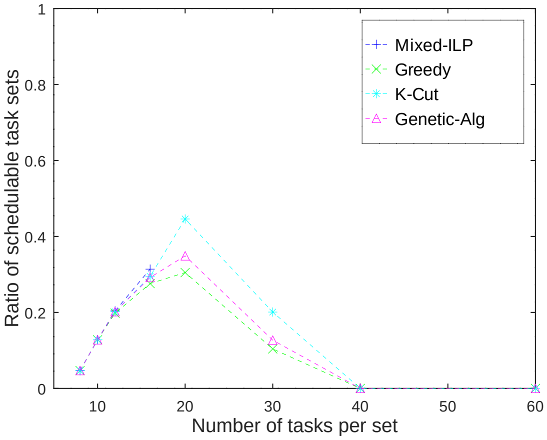

Figure 5: Schedulability ratio comparison under P-RM with varying task set size settings.

of MILP, we also presented several polynomial-time algorithms to solve this problem approximately. By conducting experiments, we evaluated the effectiveness of our model and algorithms and demonstrated our results.

Future Work. In this paper, we counted the interference and overheads of each preemption into the effective utilisation of the *preempted* task. Similar results are expected if they are counted into the *preempting* task instead, and we would like to verify this in the future. Furthermore, techniques to split such interference and overheads to be partially counted into the preempted task while allow the remaining to be counted into the preempting task were proposed in the literature [31], and we also would like to integrate those techniques in our future work. Moreover, in this paper we focus on interference-aware task-to-core mappings and assume the cache-to-core mapping has been determined *before* task partitioning process. We would like to investigate how to incorporate the cache-to-core mapping together *during* task partitioning process.

ACKNOWLEDGEMENT

We thank Ying Zhang, Lingxiang Wang, and Zhenkai Zhang for their contribution towards a preliminary version of this paper [19], specifically on finding majority of the related work, forming the ECB-UCB based ITIM calculation, and identifying an existing bound (Lemma 4.1) of the K-Cut approach. Work supported by NSF grant CNS-1850851, startup grants from University of Central Florida, and startup and REP grants from Texas State University.

REFERENCES

[1] 2012. Deadline Task Scheduling. http://https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt.

[2] 2012. Intel Platform QoS Technologies. https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies.

[3] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. 2012. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems* 48, 5 (2012), 499–526.

[4] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. 2016. On the effectiveness of cache partitioning in hard real-time systems. *Real-Time Systems* 52, 5 (2016), 598–643.

[5] Thomas Back. 1996. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford university press.

[6] Sanjoy Baruah and Theodore Baker. 2008. Schedulability analysis of global EDF. *Real-Time Systems* 38, 3 (2008), 223–235.

[7] A. Bastoni, B. Brandenburg, and J. Anderson. 2011. Is semi-partitioned scheduling practical?. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS).* 125 – 135.

[8] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. 2009. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems* 20, 4 (2009), 553–566.

[9] Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1 (2005), 129–154.

[10] Alessandro Biondi and Youcheng Sun. 2018. On the ineffectiveness of 1/m-based interference bounds in the analysis of global EDF and FIFO scheduling. *Real-Time Systems* 54, 3 (2018), 515–536.

[11] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Parallel Processing (ICPP), 2015 44th International Conference on.* IEEE, 749–758.

[12] Len Brown. 2005. ACPI in Linux. In *Linux Symposium*, Vol. 51.

[13] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. 2008. Impact of cache partitioning on multi-tasking real time embedded systems. In *14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* IEEE, 101–110.

[14] Alan Burns, Robert I. Davis, P. Wang, and Fengxiang Zhang. 2012. Partitioned EDF scheduling for multiprocessors using a C= D task splitting scheme. *Real-Time Systems* 48, 1 (2012), 3–33.

[15] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. 2000. Application-specific Memory Management for Embedded Systems Using Software-controlled Caches. In *Proceedings of the 37th Annual Design Automation Conference.* ACM, 416–419.

[16] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual, Vol.3B.*

[17] Daya Ram Gaur, Ramesh Krishnamurti, and Rajeev Kohli. 2008. The capacitated max k-cut problem. *Mathematical Programming* 115, 1 (2008), 65–72.

[18] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *Proceedings of the seventh ACM international conference on Embedded software.* ACM, 245–254.

[19] Zhishan Guo, Ying Zhang, Lingxiang Wang, and Zhenkai Zhang. 2017. Cache-Aware Partitioned EDF Scheduling for Multi-Core Real-Time Systems. In *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS) BP Session.* IEEE.

[20] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *International Workshop on Worst-Case Execution Time Analysis (WCET 2010).*

[21] Intel. 2017. Intel-CMT-CAT Pacakage. http://https://github.com/01org/intel-cmt-cat.

[22] Aamer Jaleel. 2010. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload* (2010).

[23] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT).* IEEE, 1–10.

[24] Namhoon Kim, Bryan C Ward, Micaiah Chisholm, James H. Anderson, and F Donelson Smith. 2017. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems* 53, 5 (2017), 709–759.

[25] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. 2009. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on.* IEEE, 239–248.

[26] Chang-Gun Lee, J. Hahn, Sang Lyul Min, R. Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. 1996. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *17th IEEE Real-Time Systems Symposium.* 264–274. https://doi.org/10.1109/REAL.1996.563723

[27] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* 20, 1 (1973), 46–61.

[28] Vincent Nélis, Patrick Meumeu Yomsi, Luís Miguel Pinho, José Fonseca, Marko Bertogna, Eduardo Quiñones, Roberto Vargas, and Andrea Marongiu. 2014. The challenge of time-predictability in modern many-core architectures. In *14th International Workshop on Worst-Case Execution Time Analysis.*

[29] Altmeyer Sebastian, Douma Roeland, Lunniss Will, and I. Davis Robert. 2014. Evaluation of cache partitioning for hard real-time systems. In *proceedings Euromicro Conference on Real-Time Systems (ECRTS).* 15–26.

[30] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. 2013. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Computational Science and Engineering.* IEEE, 685–692.

[31] B. Ward, A. Thekkilakattil, and J. Anderson. 2014. Optimizing Preemption-Overhead Accounting in Multiprocessor Real-Time Systems. In *22nd International Conference on Real-Time Networks and Systems (RTNS).*

[32] Y. Ye, R. West, Z. Cheng, and Y. Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT).* IEEE, 381–392.