Putting RUN into practice: implementation and evaluation

Davide Compagnin, Enrico Mezzetti and Tullio Vardanega
University of Padua, Department of Mathematics
Email: {compagnin, emezzett, tullio.vardanega}@math.unipd.it

***Abstract*—The Reduction to UNiprocessor (RUN) algorithm represents an original approach to multiprocessor scheduling that exhibits the prerogatives of both global and partitioned algorithms, without incurring the respective drawbacks. As an interesting trait, RUN promises to reduce the amount of migration interference. However, RUN has also raised some concerns on the complexity and specialization of its run-time support. To the best of our knowledge, no practical implementation and empirical evaluation of RUN have been presented yet, which is rather surprising, given its potential. In this paper we present the first solid implementation of RUN and extensively evaluate its performance against P-EDF and G-EDF, with respect to observed utilization cap, kernel overheads and inter-core interference. Our results show that RUN can be efficiently implemented on top of standard operating system primitives incurring modest overhead and interference, also supporting much higher schedulable utilization than its partitioned and global counterparts.**

I. Introduction

Compelling energy, performance and availability considerations cause the migration from relatively simple single-processor systems to more complex multiprocessor platforms. The latter in fact provide greater computational power than their predecessors without incurring as much thermal and energy drawbacks. The efficient deployment of hard and soft real-time systems on top of multiprocessor systems requires novel scheduling algorithms and analysis techniques for emerging applications to achieve high average performance and meet their real-time requirements also in the worst case. Despite the increasing research interest in the field and the relevant advances, scheduling algorithms and schedulability analyses of multiprocessor systems have not reached the same impact level as for single processors [1]. Research on multiprocessor scheduling has been primarily devoted to the main classes of *partitioned* and *global* scheduling algorithms. Partitioned approaches adopt a static mapping that assigns tasks to processors, so that each task can only be scheduled on the processor it has been assigned to. Global approaches instead do not assume any fixed task-to-processor allocation so that tasks or jobs are allowed to *migrate* from one processor to another, upon task preemptions and suspensions. None of these two classes of algorithms, however, shows decisive advantages over the other as each one has its pros and cons [1]. On the one hand, partitioned approaches, while reducing to canonical single-core scheduling problems, require solving the off-line task allocation problem, which in the general case is equivalent to the NP-hard bin-packing problem. On the other hand, global algorithms, while being naturally work-conserving and thus able to guarantee higher utilization, incur the extra overhead caused by system-wide scheduling data structures as well as the potentially destructive effects of task migration.

The fact that neither partitioned nor global scheduling offer a satisfactory general solution to the multiprocessor scheduling challenge motivates the formulation of hybrid approaches capable of attenuating both the partitioning problems and the migration overhead. This has led to the formulation of several *semi-partitioned* algorithms such as EDF-fm [2] or EDF-WM [3], where only a small subset of tasks is allowed to migrate, and *clustered* variants of global algorithms, such as C-EDF [4], where tasks are only allowed to be scheduled on (and thus migrate within) a subset of the available cores.

The *Reduction to UNiprocessor* (RUN) algorithm, first introduced in [5], broadly belongs to the semi-partitioned camp, where it brings interesting elements of originality. When applied to homogeneous multiprocessor systems, comprising m processors, RUN employs scheduling abstractions known as *primal* and *dual servers* to build an off-line data structure (*reduction tree*), which reduces the scheduling problem to m single-processor schedules, and to inform scheduling decisions at run time. The scheduling decisions in RUN are designed to minimize the number of migrations, hence the resulting overhead. RUN is claimed to be *optimal* for multiprocessor scheduling, in that it is always able to produce a feasible schedule whenever one exists. These characteristics make RUN an extremely attractive alternative to global and partitioned algorithms. However, despite having been appreciated for the elegance of the proposed solution, RUN has also given rise to concerns on its practical implementation, mainly due to the building and the representation of the reduction tree, and the overhead from frequently updating it at run time. Those doubts are best confirmed or refuted by an empirical evaluation of the algorithm. Yet, to the best of our knowledge, no such exercise has been made for RUN, nor a comparison with other algorithms.

In this paper, we provide the first solid implementation of RUN with the goal of (i) providing evidence that RUN can actually be implemented on top of standard operating system support, and (ii) producing an extensive empirical evaluation of the algorithm with respect to schedulable utilization and incurred overhead (e.g., from migration and context switches). In our implementation we use LITMUS$^{\mathrm{RT}}$ [6], [7], a real-time extension to Linux published by the University of North Carolina at Chapel Hill (UNC), which exploits a plug-in mechanism for the definition of multiprocessor scheduling algorithms.

The remainder of this paper is organised as follows: in Section II we survey the main classes of multiprocessor scheduling algorithms and present related work on their empirical evaluation; in Section III we recall the core aspects of

RUN and LITMUS$^{\mathrm{RT}}$, and introduce our implementation; in Section IV we report on the experiments we performed to assess our implementation with respect to the incurred overhead and to how the latter may affect its empirical utilization caps; in Section V we draw some conclusions and outline future work.

II. Related Work

The largest part of research on multiprocessor scheduling has focused on partitioned and global approaches. The intrinsic limitations of both classes of algorithms have been widely acknowledged [1], [8]. The main disadvantage of partitioned approaches (e.g., P-EDF) is that they must undergo a complex and inherently iterative task allocation phase whose outcome cannot guarantee total utilizations comparable to those obtainable with global algorithms. In contrast, global algorithms (e.g., G-EDF) are known to suffer from additional overhead mainly stemming from job-level migrations (not permitted by definition in partitioned approaches) and use of shared scheduling data structures. This drawback clashes against the optimality result that has been proved for the family of *proportionate fairness* algorithms, i.e., PFair [9], [10] and its derivatives (e.g., [11]). In all these algorithms the theoretical utilization bound is in practice wasted by the need for synchronization at all time slot boundaries, however small, and the ensuing migration overhead.

This scenario is evidently favourable to hybrid approaches where the critical points of partitioned and global approaches can be tempered [2], [3], [12]. However, as observed in [8], the realization of those hybrid approaches is likely to require much more complex implementations, as they retain characteristics of both global and partitioned approaches, which find no native support in existing runtimes. Hence, when reasoning on the performance of these algorithms we must pay attention to the actual overhead and implementation complexity that they incur.

Not surprisingly, the largest bulk of empirical evaluations of scheduling algorithms in multiprocessor systems has been conducted within the research group working on LITMUS$^{\mathrm{RT}}$ at UNC, where a vast collection of schedulers has been developed and evaluated [8], [13], [14]. An empirical comparison among global, partitioned and clustered EDF scheduling algorithms reported in [14] highlights the negative effects of the overhead factors in the global variant of the algorithm. The cited results also suggest that clustered variants may be a promising alternative to the partitioned one. The critical role of real-world overhead in the implementation of multiprocessor algorithms has been pointed out in [8]; in particular, that study presents a systematic description of typical overhead-related issues suffered from a number of different algorithms, and reports the empirical overheads incurred by a number of variants of EDF and the Notional Processor Scheduling - Fractional capacity (NPS-F). The work reported in [13] instead provides an empirical evaluation of several scheduling algorithms (accounting for scheduling overheads) running on top of a multicore Niagara platform.

The first attempt at implementing RUN, also using LITMUS$^{\mathrm{RT}}$, has been reported in [15]. The authors of that work take the view that the implicit-deadline periodic nature of the task sets amenable to RUN is served well by a static scheduling table. Though plausible, we found that choice to be rigid and unfit to scale to non-harmonic task sets, where extremely large hyperperiods might appear, resulting in inordinate growth of the table size. In our implementation, we opted instead for the original formulation of RUN, with dynamic scheduling based on an actual reduction tree

III. The RUN Scheduler

Before the advent of RUN, optimality for multiprocessor scheduling was proved for the family of algorithms based on *proportionate fairness*, whose progenitor was PFair [9], [10]. RUN aims at mitigating the massive overhead incurred by those methods while preserving their optimality. It is worth noting that another optimal algorithm, namely U-EDF [16], does indeed break the principle of proportionate fairness. The main claim of RUN, however, is that a multiprocessor scheduling problem can be reduced to a series of uniprocessor problems, for which good scheduling algorithms (e.g., EDF) exist, without need for partitioning.

In the following, we present our implementation of RUN on top of LITMUS$^{\mathrm{RT}}$. Before delving in the details, we provide a short description of RUN in its original formulation [5], and a brief introduction to the LITMUS$^{\mathrm{RT}}$ environment.

A. The RUN algorithm

In RUN, a system is composed of n independent real-time tasks, each generating an infinite sequence of jobs J, everyone of which characterized by a release instant $J.r$, a worst-case execution time (WCET) $J.c$ and a deadline $J.d$, executing on top of m homogeneous processors. In the intent of representing possibly non-periodic execution, RUN branches off from the canonical sporadic task model and introduces the *fixed-rate* task model. Under that model, tasks are characterized by a constant execution rate $\mu \leq 1$ and an infinite set of deadlines D. A fixed-rate task τ_i can thus be referred to as a pair (μ_i, D_i). A rate in fact represents the execution requirements of a task in terms of a fraction of processor utilization with respect to any interval determined by a job release and the corresponding deadline. Therefore, for any job J of a fixed-rate task τ_i it holds that $J.r \in \{0 \cup D_i\}$, $J.d = min_d\{d \in D_i, d > J.r\}$ and $J.c = \mu_i(J.d - J.r)$. This characterization applies to both tasks and groups thereof, for which aggregate rate and deadlines are computed respectively as the sum and the union of the individual counterparts. The concept of fixed-rate tasks is fundamental to the formulation of the algorithm as it eases combining execution requirements of multiple tasks when they are grouped together into a server. The definition of fixed-rate task given in [5] inaccurately stretches to *non-periodic* tasks: in actual fact, the definition of fixed-rate task only fits periodic tasks with implicit deadlines, and their aggregates or servers.

The reduction process in RUN builds on the notions of *dual scheduling* and *packing*. Dual scheduling, first introduced in [9], leverages the observation that solving the problem of scheduling a given task set $\mathcal{T} \ni \{\tau_i\}_{i\in\{1,n\}}$ with utilization U, is equivalent to scheduling the *dual* task set $\mathcal{T}^*$, where the latter is composed of tasks τ_i^* with exactly the same period and deadline of τ_i, but complementary utilization u_i^*. On this basis, it has been proved in [5] that the total utilization U^* of the dual task set $\mathcal{T}^*$ is equal to $n - U$: on this account whenever $U^* = n - U < m$, it is convenient to solve the problem of scheduling $\mathcal{T}^*$ on a reduced number of $\lceil U^* \rceil$ processors. In particular, to

enforce $n - U < m$ it is sufficient to guarantee n to be small. The latter property is achieved in RUN by packing tasks into scheduling abstractions, termed *servers*, and later recursively packing servers into higher-level servers. A server S is therefore a scheduling abstraction serving as a proxy for a collection of "client" tasks $\mathcal{T}$ from which it inherits the set of deadlines and rates. Similarly to fixed-rate tasks, S is characterized by a pair (μ_S, D_S) where $\mu_S = \sum_{\tau_i \in \mathcal{T}} \mu_i$ and $D_S = \bigcup_{\tau_i \in \mathcal{T}} D_i$.

By alternately packing tasks (servers) and reformulating the scheduling problem into its dual, the number of processors is progressively reduced until it reaches 1 [5], thereby reducing the initial problem to a set of uniprocessor scheduling problems. The entire reduction is performed off-line and each reduction step corresponds to a reduction level in a path starting from the original tasks ending up in the final *unit* server that can be scheduled on a uniprocessor. Reversing this path, we obtain the so-called *reduction tree* which is exactly the data structure that is used to derive scheduling decisions at run time.

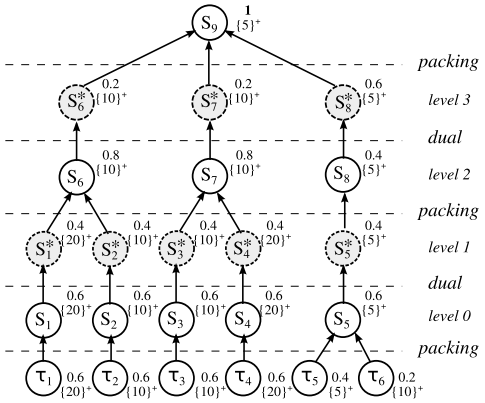


Fig. 1. Example of RUN reduction.

Figure 1 shows an example of reduction and the resulting reduction tree for a simple task set comprising 6 tasks. Each task and server is characterized by rate and set of deadlines (within brackets). A $\{D\}^+$ notation is used to indicate that a set of deadlines D is infinitely replicated over $\mathbb{N}$. Reduction proceeds in a bottom-up fashion. Tasks are initially packed into level-0 servers so long as their individual cumulative utilization does not exceed 1: for example τ_5 and τ_6 are packed together into S_5, with a cumulative rate of 0.6 over an enlarged set of deadlines $\{5,10\}^+$ ($\{5\}^+$ in short). Instead τ_1, τ_2, τ_3 and τ_4 cannot be involved in any packing. The obtained servers are new scheduling entities whose characteristics (i.e., rate and deadline) directly derive from their constituents: servers thus correspond to virtual tasks generating *virtual jobs*. The next step consists in switching to the dual representation of the problem, which yields the respective set of dual servers S_i^*. Each dual server exhibits a complementary rate but inherits the same set of deadlines of its primal counterpart. For example S_5^* in Figure 1 is characterized by a rate of $1 - \mu(S_5) = 0.4$ and deadlines $\{5\}^+$. This sequence of steps is then repeated until we get a set of servers S_6^*, S_7^*, S_8^* with a total utilization 1, fit for single-processor optimal scheduling. By packing the last level of dual servers into a *unit* server S_9 we set down the root of the reduction tree, whose leaves regroup the original task set. Under the assumption of full system utilization, a sequential application of the dual and pack operations is always guaranteed to converge to a set of unit servers.

At run time, scheduling is performed according to the information in the reduction tree, which is dynamically updated at each virtual job release and completion. On a tree update, task selection is done by applying two simple rules at each level l of the tree. As a first rule, each primal node at level l selected for execution (*circled* in RUN speak) at time t shall select for execution (i.e., circle) its dual child node at level l-1 with the earliest deadline; according to the second rule, each node at level l-1 which is not circled in the dual schedule is selected for execution in the primal level l-2 and, vice-versa, each node which is circled in the dual level l-1 is not circled (i.e., kept idle) in the primal l-2. Applying these rules from the root downwards will lead to the selection of those tasks that must execute. The correctness of the relationship between primal and dual schedule is guaranteed by RUN's full-utilization assumption and its fixed-rate model requirements.

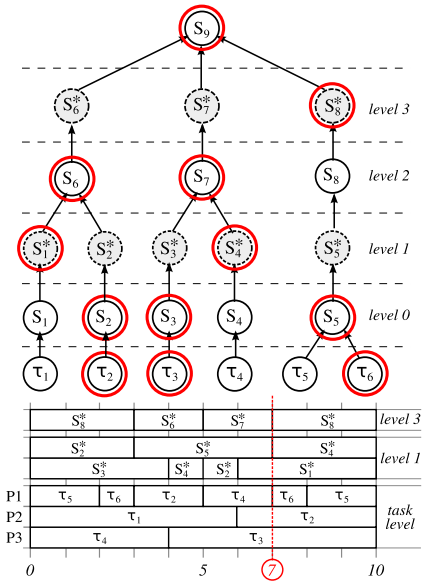


Fig. 2. Example of RUN scheduling.

Figure 2 shows a possible task selection as determined by the state at time t=7 of the reduction tree depicted in Figure 1. In this case, we observe a level-3 switch between servers S_7^* and S_8^* owing to the termination of a S_7^* virtual job. Following the RUN scheduling rules, this event triggers a tree update that results in a level-0 switch between S_4 and S_5, which in turn causes a single task-level switch between τ_4 and τ_6. Eventually tasks τ_2, τ_3 and τ_6 are selected for execution. In contrast with G-EDF, RUN is prevented from making greedy scheduling decisions by the proportionate execution shares that are guaranteed by *virtually scheduling* servers. A job of a server, in fact, represents an execution *budget* (proportional to the server rate) allocated to the server children. Therefore scheduling within each interval I is done by selecting the nodes in the reduction tree according to the RUN rules, and replenishing their budget in I.

The RUN's authors classified it within the camp of semi-partitioned algorithms. In fact, RUN is not a semi-partitioned approach in the common acceptation of the term; it rather falls into a separate category, which can be called hybrid, as it may behave more or less similarly to a global or partitioned algorithm. The packing step does pair RUN with partitioned approaches, but not exactly since tasks are not associated to

processors but to servers, which in turn are not pinned to any particular processor as in global scheduling.

The approach in RUN is global in the sense that scheduling decisions are taken globally and the packing step is much less critical to the final performance of the algorithm as compared to, for example, P-EDF. The reduction process in RUN is not as crucial as the packing step in partitioned algorithms as it may influence the shape of the reduction tree, but it cannot compromise the validity of a schedule (which P-EDF packing instead does). Yet, with favourable task sets, the reduction tree may produce exactly the same behaviour as P-EDF.

Behaving as partitioned or as global depending on the task set is one attractive trait of this algorithm. As observed in [17], what is actually partitioned is the concept of proportionate fairness, which is proportionate to servers (i.e., group of tasks).

B. *The* LITMUS$^{\mathrm{RT}}$ *environment*

The LInux Testbed for MUltiprocessor Scheduling in Real-Time systems is an extension to the Linux kernel (at version 2013.1 for this paper) developed by UNC for the experimental development and evaluation of scheduling algorithms and locking protocols in multiprocessor systems.

When it comes to scheduling, LITMUS$^{\mathrm{RT}}$provides numerous utility components (e.g., queues, timers, etc.), an augmented set of system calls for real-time tasks, and a simple plugin interface to activate a specific scheduling policy at run time. These features enable the definition of a large class of user-defined scheduling algorithms, some of which have been in fact already developed and made public along with LITMUS$^{\mathrm{RT}}$ releases [7]. Implemented algorithms include global, partitioned and clustered variants of EDF as well as an implementation of the partitioned fixed-priority (P-FP) and PFair algorithms. Moreover, LITMUS$^{\mathrm{RT}}$ offers a score of interesting tracing and debugging capabilities that form an extensively automated framework for the development and evaluation of scheduling primitives and kernel overheads. In our case, the distinct similarity of treating global release events suggested that the LITMUS$^{\mathrm{RT}}$ implementation of G-EDF was a good base for our implementation of RUN.

C. *The RUN plug-in*

The description of RUN in [5], [17], though profound in theory, leaves ample room for design decisions and optimizations. In the following, we present our implementation of the algorithm[1] and discuss the practical issues we encountered. We start from the description of the data structure adopted to represent the reduction tree.

1) Reduction tree data structure: Although no strict requirement is set by RUN on the way a reduction tree should be implemented, performance considerations suggest adopting simple and compact data structures. We therefore focused on devising a convenient representation for the reduction tree such that its attributes could be efficiently updated at run time with fresh deadline and budget information [5].

Our intuition was that the reduction tree is somehow redundant at run time, in that it provides both the primal and the dual representation of every single server, where one of them suffices. We therefore opted for a simple compact representation based on a *left-child right-sibling* binary tree [18] to model just the dual nodes of the original tree. With RUN, at each release event occurring at level-0, all deadlines along the path to the root of the reduction tree must be updated: to speed the bottom-up traversal of the tree, each node is augmented with a backward link to its parent node. Figure 3 hints at the implemented data structures: beside the navigation links, each node in the tree holds the information needed to take scheduling decisions: the earliest deadline among the children nodes; the current (updated) budget; and a flag indicating whether the node is currently circled for execution.

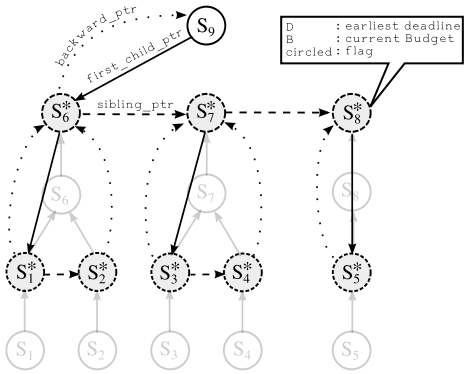


Fig. 3. Reduction tree data structure.

Each node in the tree is also assigned an interval timer, to keep track of when the execution budget attached to it is consumed. Whenever a node selects a child node for execution, it also arms its timer according to the spare budget of the selected node. Thus, a timer expiration event corresponds to a *budget exhaustion* on a child node and triggers a tree update, which in turn causes the system to be rescheduled. From the scheduling standpoint, a budget exhaustion corresponds to a completion of a server job (a *virtual* event in [5]). To discriminate between virtual and real events we use the term budget exhaustion to address completion events on servers at level $k > 0$. In contrast with budget exhaustions, completion events are local events to the processor where the completed job was executing: no tree update and global reschedule is required in that case.

Admittedly, our representation of the reduction tree as a binary structure is somewhat adverse to EDF scheduling. With sibling nodes organized as a linked list, in fact, finding and selecting the child node with earliest deadline requires traversing the whole list. To that effect, an ordered heap would probably earn better performance. However we should also note that the scheduling tree is constantly shared among all processors and not pinned to a specific one, as scheduling decisions are taken on any core. This means that the reduction tree may notionally *migrate* from one processor to another: in that respect, a compact data structure is clearly preferable.

In our implementation of the algorithm we did not pay particular attention to how tasks and servers are packed together to form the reduction tree: we simply opted for worst-fit packing, similarly to [5]. It is important to appreciate however that different packing policies may lead to different reduction trees,

[1]Our RUN plug-in is available at http://www.math.unipd.it/~dcompagn.

which may result in different preemption and migration patterns at run time. There is room for some fine tuning in this regard.

2) Scheduling on the reduction tree: The on-line part of the algorithm relies on having the reduction tree always up to date with the deadline and budget information. Updating the tree is perhaps the most critical operation in RUN as, in order to keep that information consistent, it must be executed at each level-0 job release. Since the nodes in our tree just hold the information on the next deadline, we only update the nodes along the path to the root if the release event causes the adjustment of the current sub-tree interval.

The application of RUN rules to the tree for the purpose of scheduling is done not only on release events, but also at budget exhaustion. With our implementation of the reduction tree, the latter event corresponds to the expiration of a timer associated to a node *at any depth of the tree*. In that event however, we need not reapply RUN scheduling to the whole tree: instead, we may restrict our attention to the sub-tree rooted at the node where the exhaustion event occurred. This shortcut stems from the observation that a context switch at any level of the server tree is guaranteed to trigger exactly one context switch in the underlying level, as shown in [5] (Lemma V.4). Assume node S, at some level of the tree, detects a budget exhaustion on behalf of one of its children. Then S selects its next child and triggers a context switch whose effects are propagated downward to determine a context switch between exactly two leaf nodes in the sub-tree rooted in S.

Performing scheduling operations on a multiprocessor involves using either local or global locks. From that perspective RUN is inherently a global scheduling algorithm and, similarly to G-EDF, is implemented with global locks. This notwithstanding, some scheduling decisions in RUN are inherently local: for example, dispatching triggered in response to a job completion event. In those cases, we must consider that the corresponding server may be incidentally evicted from the processor: the local scheduling structures must therefore be suitably protected. RUN is silent on the actual assignment of tasks to processors, except recommending unnecessary job migrations should be avoided. Between all the available processors, we should preferably select the one we were executing before being preempted. In practice, however, it is rather infrequent to have more than one potential processor available: this circumstance depends on the depth of the reduction tree.

3) Overlapping events: Our implementation of RUN does not prevent global events from coinciding, be they job releases or budget exhaustion events. In fact, job releases and exhaustion events stemming from separate sub-trees in the reduction tree may overlap as a result of (i) simultaneous releases, (ii) sequential budget exhaustion and replenishment on the same node, or (iii) simultaneous budget exhaustion. Overlapping events may cause some difficulty as the global lock contention involved in handling them may cause unnecessary sequences of tree updates, and thus incur considerable overhead.

Simultaneous job release from different servers does not pose a problem to our implementation as the release events are automatically collapsed by having merged each server release queue into a single global release queue. More challenging is the case of a budget exhaustion event on a node S_i that overlaps with a budget replenishment on the same node, as a result of a new job being released at a lower level in the tree. To prevent inconsistent behaviour, we avoid arming a budget exhaustion event if it overlaps with a release event as the tree will be consistently updated anyway upon the release event. Simultaneous budget exhaustion events, instead, cannot be easily merged together as they belong to different sub-trees and cannot be intercepted locally without incurring major overheads.

4) Job migrations: RUN's highlight is its promise of cutting on the number of preemptions and migrations suffered, in comparison to optimal scheduling algorithms that seek proportionate fairness. It can be seen however, that the extent of saving strongly depends on how tasks are packed into servers, as different reduction trees may result in different preemption and migration patterns at run time. How to address that dependence without backfiring on application design or tentatively exploring off-line heuristics is an open problem, which we are currently studying. Our current implementation does not strive to produce "good" packing and contents itself with worst-fit.

It is worth noting that RUN only allows push-based migrations where the processor that should schedule a job is statically determined, similarly to standard partitioned or semi-partitioned scheduling algorithms. As observed in [8], this kind of migrations incurs considerably less overhead than dynamically computed migrations, which use global data structures and the corresponding locks.

5) System workload: Our implementation had to figure how to deal with the full utilization requirement set by RUN, which is prerequisite for duality to exist and thus for the correctness of the algorithm. This requirement, in fact, is evidently unrealistic in practice: the use of forced idle times or dummy tasks is suggested in [5] to cope with (i) variable job execution times and (ii) low utilization. With respect to (i) we simply exploit the structure of LITMUS$^{\mathrm{RT}}$ that allows native tasks to execute when no *litmus* task (part of the RUN schedule) is ready. Within the scheduling domain of a level-0 EDF server, if a job J completes its execution earlier than expected (i.e., its actual execution request is less than the assumed WCET), then we either execute the next ready EDF job in the server, if any, or do just nothing, which allows the execution of native tasks. Advancing in the local schedule underneath LITMUS$^{\mathrm{RT}}$, in fact, does not impair the overall RUN schedule.

When it comes to (ii), instead of further increasing the task population, we preferred to turn the slack time into a buffer for run-time kernel overheads, which [5] does not contemplate. We therefore assigned balanced shares of slack time to each level-0 server, possibly obtaining new unit servers. This way our implementation can correctly operate for applications with utilization under 100% while also being free from the risk of occasional overruns[2]. Incidentally, it is worth noting that deciding slack allocation has much to do with building the reduction tree and consequently with the intent of containing the number of preemptions and migrations that may occur at run time. Regardless of the overall utilization, in case of extremely low population, when there are less tasks than processors (i.e., $n < m$), the reduction has to be conducted as if there were just n processors.

[2]In RUN, a deadline miss causes inconsistencies in the reduction tree and therefore wreaks the system state.

D. LITMUS^{RT}-related considerations

Although LITMUS^{RT}provides several powerful components and functionalities that facilitate the plugin development, the peculiarities of RUN posed some interesting challenges.

1) Scheduling domains: LITMUS^{RT}offers `rt_domain` as a powerful abstraction for the realization of real-time scheduling domain. Support for it comes with a generic and reusable implementation of a scheduling context, comprising task queues and timers. The structure of the reduction tree (see Figure 2) and the implied hierarchy of run-time entities suggests associating an `rt_domain` to each server (either primal or dual): this way each server would have its separate scheduling events and queues. In fact, each node in the tree should be able to schedule its own children independently, just following EDF rules.

Unfortunately, this solution is unable to collapse simultaneous events into the same handler. The simultaneous release of k jobs, for example, would be potentially triggered by k release events and the strictly sequential acknowledgement of each such events – which involves updating the whole tree – would incur an inordinate amount of overhead. The opposite approach, with a single global `rt_domain` for the whole tree still fails to distinguish between global and local events. In contrast with a job release, a job completion is not necessarily a global event but may need to be handled locally by its parent node (i.e., level-0 server). In addition, using global system-wide scheduling structures leads to the overhead typical of global scheduling algorithms, which we want to avoid.

We opted for a middle-ground solution where only inherently global events are handled globally and local events and scheduling structures are kept local within each server. Instead of defining a custom version of basic scheduler components for task queues and timers, we employed the `rt_domain` in a non-standard fashion. We used a global `rt_domain` just to handle all release events: no information is stored in its task queues. A local `rt_domain`, instead, is associated to each level-0 server, to handle its local ready queue. The global `rt_domain` allows optimizing and collapsing simultaneous release events (whose effects are global in RUN), whereas the local ones contain the scope of the scheduling data structures.

2) Off-line reduction phase: Although the construction of the reduction tree seems quite similar to the preliminary packing phase of partitioned algorithms, its actual implementation may raise some issues. The main difference between populating the tree in RUN and filling the bins in (semi-) partitioned approaches is that servers, in the former, are an abstract concept whereas bins, in the latter, correspond to actual processors. As a consequence of detaching the `rt_domain` from a concrete processor, tasks and their queues are not directly associated to a processor and no task can be created until the respective server (in the tree) has been defined (in the system). A serialization of the tree is given in input to the plugin and then reconstructed in the kernel in a sort of a pre-initialization phase, before admitting tasks in the system.

IV. Evaluation

An extensive simulation-based evaluation of RUN has been reported in [5], [17]. The cited works show that RUN significantly outperforms other optimal multiprocessor scheduling algorithms in terms of incurred preemptions and migrations. As observed in [19], the evaluation of a scheduling algorithm by simulation may not be always exhaustive: however accurate, it cannot be as complete as the actual implementation. Particularly for scheduling algorithms, empirical evaluations may unveil unexpected implementation challenges and exhibit actual kernel overheads that may not always be evident. The proposed implementation of RUN on top of LITMUS^{RT} demonstrates that the algorithm can be implemented on standard operating system support, and enables extensive empirical evaluations of it.

In our experiments we were interested in evaluating (our implementation of) RUN from the perspective of the kernel overhead, captured in terms of the cost of its scheduling primitives, and the number of incurred preemptions and migrations. We did not focus on finding the empirical utilization cap of our implementation, for comparison with other optimal scheduling algorithms. We therefore do not evaluate RUN against S-PD^2 [13], an efficient implementation of PFair [10], included in LITMUS^{RT}, though it is shown to have competitive scheduling overheads, low bus contention, and decent schedulable utilization. Our rationale is that, as PFair and S-PD^2 fall in the category of quantum-driven scheduling, they are inherently exposed to larger preemption and migration overheads than event-based solutions[3]. In future work, we will return to evaluating RUN from the angle of optimality. To that end, we regard U-EDF [16] as an interesting term of comparison in that, similarly to RUN, it does not build on proportionate fairness. In the experiments presented here, we compare RUN against P-EDF[4] and G-EDF (both included in the LITMUS^{RT}2013.1 release) as representatives of global and partitioned algorithms, respectively. The choice of P-EDF and G-EDF is not arbitrary as: (i) RUN reduces to the former when the first *pack* operation produces as many unit servers as processors; and (ii) global scheduling events in RUN are expected to incur overheads that are comparable to those observed for G-EDF.

A. Experimental setting

The experiments consisted in collecting execution traces obtained by feeding identical task sets to the three scheduling algorithms and then comparing the respective execution statistics. It stands to reason that the performance of each algorithm depends on the task set features and the overall system workload in particular. To warrant fair evaluation, we focused on randomly generated task sets with increasing overall system utilization, between 50% and 100%. The granularity of our observations increases while we approach higher system utilization as we expect most important variations to occur then. When it comes to the task population, we wanted our task sets to be equally representative of light and heavy tasks. To this end, task utilizations were generated following a bimodal distribution over the two intervals [0.001,0.5) and [0.5, 0.9], with probabilities respectively equal to 45% and 55%. We are also aware that the performance of some algorithms may be affected by the distribution of task periods, which in turn affects the distribution of release events. For this reason, we

[3]We performed some experiments to sustain our claim. In them, which are available at http://www.math.unipd.it/~dcompagn, S-PD^2 incurred up to six times the number of preemptions and migrations observed in our RUN implementation.

[4]The same *worst-fit* bin-packing heuristic was applied for both P-EDF and RUN to allow a fair comparison.

performed two sets of experiments targeting harmonic and non-harmonic task sets. In both cases, tasks periods were drawn from a uniform distribution in [10ms; 200ms] so as to keep the experiment manageable in size and complexity. We used the lightweight tracing facilities offered by LITMUS$^{\mathrm{RT}}$ to collect execution traces for the three contenders, where each task set was executed for 30 seconds spanning multiple hyperperiods.

The determination of the kernel overhead is a fundamental issue in any empirical study of scheduling algorithms in that it is at the same time a target and a prerequisite of the evaluation. It is also a prerequisite because, under some circumstances (e.g., zero-laxity condition and higher system utilizations), disregarding the kernel interference may compromise schedulability. For this reason, we organized our experiments in two phases. A first round of experiments was devoted to the measurement of the overheads incurred by each scheduling primitive. The collected information was then used in the second phase to determine an upper-bound for the global scheduling overheads and to account for them in the execution time of each task.

Experiments were run on a dual-processor AMD Opteron™ 2356 system where each processor includes 4 cores running at 2.3 GHz with private L1 and L2 caches of 64 and 512 KB respectively. Power scaling and power management functions were intentionally disabled.

B. Accounting for kernel overheads

As observed in [8] many different factors contribute to the scheduling overhead for a given algorithm in multiprocessor systems. In order to get trustworthy figures for the cost of each scheduling primitive we used the LITMUS$^{\mathrm{RT}}$ tracing facilities to collect the execution traces of more than 600 randomly generated task sets, with increasing total utilization. From this large set of experiments we were able to derive an empirical upper bound to the execution cost of each scheduling primitives for each algorithm involved in our comparison. This knowledge was then used in the dimensioning of the second round of experiments.

Preliminary information on the system overheads would have been required to have full control on the "real" system utilization. However, this first round of experiments served specifically to gauge the impact of such overheads and could not exploit any prior knowledge. As a workaround, and only for this first round of observations, we excluded task sets with total utilization larger than 95%, so that a residual 5% system utilization could be reserved to tracing and scheduling overheads.

The overall system overhead in multiprocessor systems is determined by several factors [14]: (i) the *release* (REL) of a new job, (ii) a *scheduling* (SCHED) event and (iii) the induced *context switch* (CSW), (iv) the *interrupt latency* (LAT), including inter-processor interrupts (IPI), and (v) the *tick interference* (CLK). In addition to those factors, RUN incurs an additional source of overhead originating from the run-time *update* (TUP) of the reduction tree, in response to a job release or a budget exhaustion event. Figure 4 reports the maximum cost (in μs) we observed for each individual overhead factor under each scheduling algorithm. These numbers confirmed our expectations: whereas all three algorithms exhibit similar behaviour with respect to context switch and tick interference, P-EDF outperforms both RUN and G-EDF with respect to the cost of the job release and schedule primitives. This behaviour is easily explained by the fact that P-EDF does not contend for global locks and typically operates on smaller scheduling structures. Interestingly, the worst-case costs here reported were collected from the observations on harmonic task sets where many overlapping events (e.g., simultaneous release) are handled within the same kernel primitive.

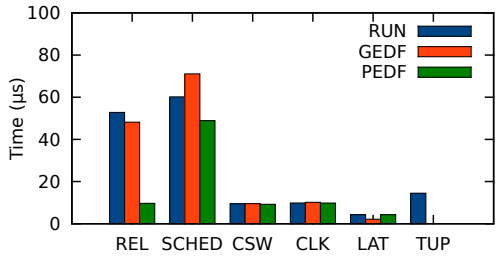


Fig. 4. Maximum observed system overheads.

The release (REL) primitive for RUN also includes the update of the reduction tree (TUP isolated on the rightmost bar in Figure 4): for this reason its worst-case cost is even worse than for G-EDF. Further indirect sources of interference are incurred upon *preemption* (PRE) and *migration* (MIG) as they may affect the execution context of a job (e.g., its cache state).

The scheduling overhead is commonly accounted for by augmenting each task WCET so that the execution time of each job also includes the scheduling overheads it may cause (or incur) [20], [21]. We adopted this same approach within the LITMUS$^{\mathrm{RT}}$ framework by exploiting a low-level scaling mechanism available for `rtspin`, a dummy real-time task skeleton we used as a baseline for our experiments. Deflecting from its original intent, we used that mechanism to remodulate the WCET of each task to account for a conservative per-job overhead bound. The actual overhead suffered by a job, however, strictly depends on the the scheduling algorithm.

Our focus in this study was set on finding an empirical upper bound for the scheduling overhead incurred by RUN. Although we effectively measured the cost of all kernel primitives for all contender algorithms, we took RUN as a reference for upper-bounding the scheduling overhead potentially suffered by each job. Each job in RUN suffers a score of overheads in its life cycle: release overhead when it becomes ready, plus scheduling; IPI latency and context switch overheads when it gets dispatched. Moreover, a job release can cause some other jobs to be preempted: in particular the original authors of the algorithm showed in [5] that the number of preemptions caused directly by a job or by its ancestors (servers) depends on the height of the reduction tree; in the worst case, it is at most $k = \lceil (3p+1)/2 \rceil$, where p is the tree height. Therefore, in average, each job may suffer at most k preemptions, which is a relatively nice bound as the tree height rarely exceeds 3 in practice. In case of preemption, the suffered overhead is similar to that incurred upon release with the only difference that a TUP overhead has to be paid instead of the REL one and a PRE or MIG overhead has to be accounted. Accordingly, we computed the system overhead for RUN as a simple additive formula:

$$OH^{Job}_{RUN} = REL + \widehat{SCHED} + CLK + \\ + k \times (TUP + \widehat{SCHED} + max(PRE, MIG))$$

where $\widehat{SCHED} = SCHED + CSW + LAT$.

With respect to the cost of preemptions and migrations, we understand that both are largely determined by cache-related interference [8], [22] and are therefore difficult to determine. However, in our setting, cache effects are not so critical because all tasks are synthetically equivalent and feature extremely reduced working sets. Hence the PRE and MIG overheads we accounted for in our bound were determined empirically by ad hoc observations[5] and are not truly representative of cache-related effects. We defer a detailed quantification of those overheads to future work. An extensive evaluation of preemption and migration overheads for global and partitioned EDF has been reported in [14].

It should be noted that the computed overhead OH^{Job}_{RUN} overestimated the average overhead observed per-job for all three algorithms. The OH^{Job}_{RUN} bound was then used in the second phase of our experiments to determine in LITMUS$^{\mathrm{RT}}$ the share of each task's WCET to be reserved as a buffer for system overheads.

C. Results

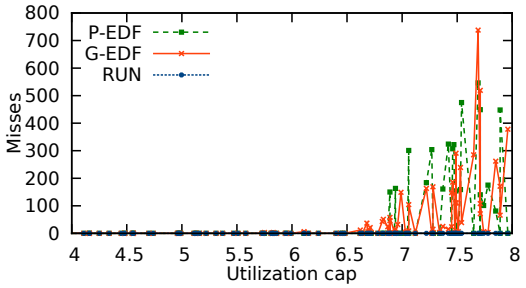
Figure 5 reports the empirical schedulability results we observed for the three algorithms under increasing overall system utilization (on the horizontal axis). Note that since we executed our experiments on an 8-core target, the extreme values on the axis stand for 50% and 100% utilizations respectively. Figure 5a on the left shows the cumulative misses we observed in our experiments: whereas both G-EDF and P-EDF incur an increasing number of misses with higher utilizations, RUN incurs no misses. Not surprisingly, RUN proves capable of sustaining extremely high workloads, up to a theoretical 100% utilization. As shown in Figure 5b, reporting the percentage of infeasible task sets (i.e., for which we observed at least one deadline miss), P-EDF is penalized by higher utilizations as it cannot guarantee good solutions to the packing problem (cured instead by dual representations in RUN). G-EDF, instead, already produces invalid schedules at medium utilization levels, possibly penalized by greedy scheduling decisions. Besides optimality considerations, the fact that we actually observed no misses for our RUN implementation is also explained by the way we accounted for the system overhead, which was explicitly tailored to RUN. Although Figure 5 reports the statistics for the non-harmonic task sets, it is worth noting that we observed the same trend also in the harmonic setting. This seems to suggest that in our experiments deadline misses are correlated to the limitations of each individual algorithm rather than to the incurred scheduling overheads.

A major objective of our experiments was to assess the overhead induced by RUN with the intent of assessing its actual applicability. An empirical evaluation of the kernel interference, in terms of observed preemption and migration events (still ordered by increasing system utilization) is shown in Figure 6. Figure 6a reveals that in our experiments RUN incurred exactly the same number of preemptions as P-EDF, within the empirical feasibility region of the latter (i.e., $U \leq 6.5$). This is explained by the fact that in most cases the packing heuristic (used by both P-EDF and RUN) was able to find a perfect packing: in those cases RUN reduces to P-EDF, modulo the slack required to meet the 100% utilization requirement. Excluding its invalid schedules, G-EDF incurs only slightly fewer preemptions than RUN; however, almost all such preemptions result in a job migration, as shown in Figures 6b and 6d. On the contrary, RUN only incurs some migrations at higher utilizations ($U \geq 6.5$) where it features non-trivial reduction trees (Figure 6c). RUN encounters more migrations than G-EDF only with $U \geq 7.5$, where the latter produces invalid schedules: this could be seen as the price to be paid to guarantee task set feasibility.

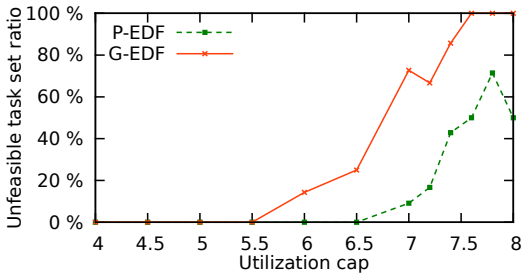
Figure 7 focuses on core scheduling primitives and shows the sheer cost of the release and dispatching events under RUN, P-EDF and G-EDF. We report here the average costs under increasing system utilization, whereas we already discussed the worst-case observed costs in Section IV-B. As expected, the use of local (per partition) data structures allows P-EDF to outperform the other algorithms relatively to the cost of a job release. At lower utilizations, we observe that job releases are slightly more onerous in RUN than in G-EDF. This is explained by the fact that the release primitive in RUN always includes the cost of updating the reduction tree (TUP in Figure 4) which is indeed comparatively onerous, especially in case of small task sets. Arguably, if we exclude this contribution, RUN always outperforms G-EDF. The cost of the schedule primitive is much more critical as it is more frequently invoked than the release primitive (i.e. once per job). In this case, our implementation of RUN does not rely on a global ready queue, potential source of contention in G-EDF, but uses a separate queue for each level-0 server. With increasing system utilizations, the cost of a schedule in RUN is thus more similar to that of P-EDF.

For the sake of completeness we recall that the tree update primitive is also triggered as a result of a budget exhaustion event. However, consistently with the claim in [5], our experiments never needed trees with more than 2 levels: under these conditions the induced overhead is reasonably low. This observation is confirmed by Figures 8a and 8b, showing the average *per-job* scheduling overhead for harmonic and non-harmonic task sets respectively. The overhead reported in these diagrams includes the cost of all kernel primitives (thus all tree updates in RUN) plus the cost of the other system overheads. With harmonic task sets (Figure 8a), the average overhead incurred by the three algorithms is quite stable: in this setting, the RUN behaviour is comparable to P-EDF and never worse than G-EDF. With non-harmonic task sets (Figure 8b) the average overhead is imperceptibly reduced due to slightly less onerous scheduling primitives (as explained in Section IV-B). This trend, however, is less apparent for RUN: while still exhibiting a modest per-job overhead, RUN suffers from the fact that tree updates are more frequently triggered in non-harmonic task sets. This is even more evident for utilization $U \geq 7.2$ where we observed a steep increase of the per-job overhead. This behaviour cannot be explained exclusively by observing that at higher utilizations RUN requires more reduction levels to find a proper reduction: as shown in Figures 8c and 8d, we experimented a similar increase in the average number of levels in the reduction trees also in the harmonic experiments. In the non-harmonic setting, in fact, the growth of the reduction tree also entails an increase in the number of "virtual" scheduling events (i.e., budget exhaustions) and their incurred cost (i.e., to update the reduction tree). This hypothesis is confirmed by the average preemptions per job, reported in Figures 8f and 8f, where non-harmonic task sets incur almost twice as many

[5]The MIG overhead always upper-bounded the PRE one in our experiments.

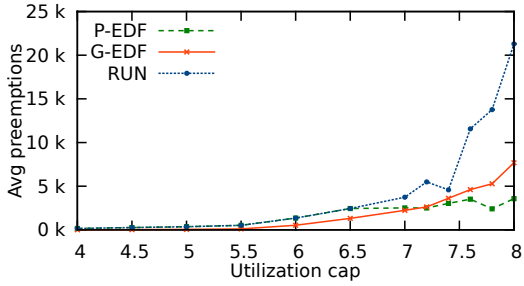


(a) Observed misses.

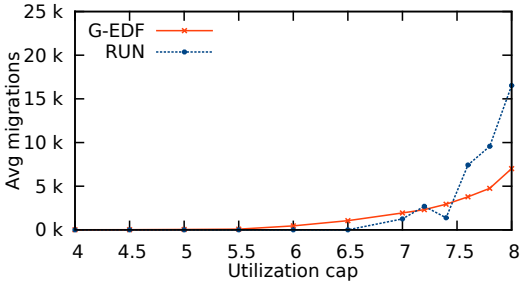


(b) Infeasible task sets ratio.

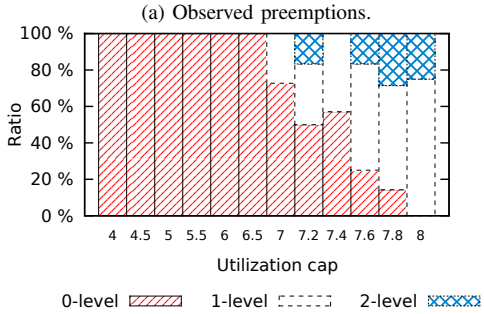
Fig. 5. Empirical feasibility.



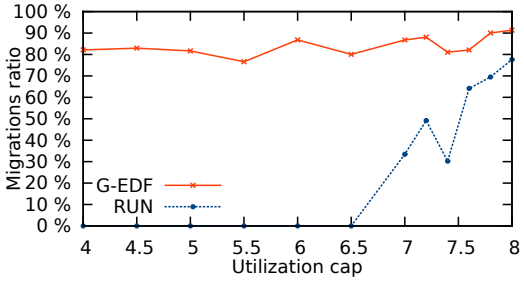
(a) Observed preemptions.



(b) Observed migrations.

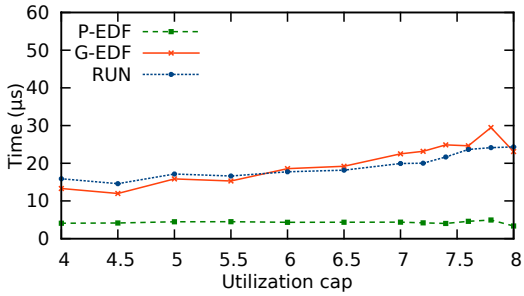


(c) Tree levels per utilization.

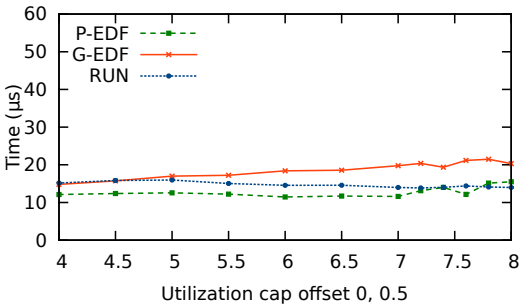


(d) Migration ratio.

Fig. 6. Kernel interference.



(a) Average release overhead.



(b) Average schedule overhead.

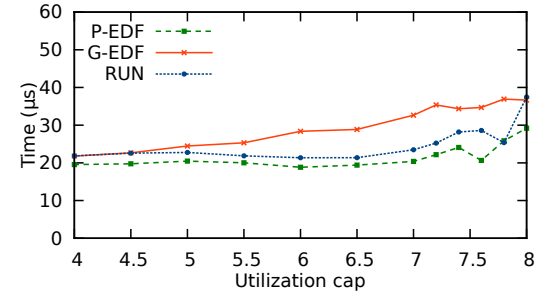
Fig. 7. Release and schedule primitives.

preemptions per job in the average case.

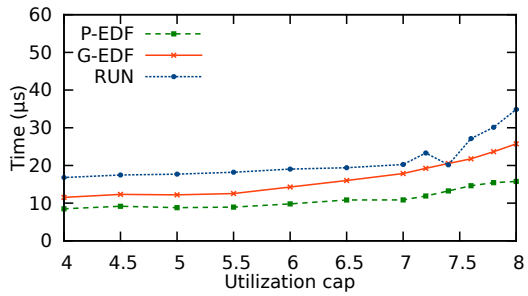
V. Conclusion

We presented in this paper the first solid implementation of the RUN multiprocessor scheduling algorithm on top of the LITMUS$^{\mathrm{RT}}$ testbed. We demonstrated that RUN can be easily implemented on standard operating system support. We also presented an empirical evaluation of RUN that disproves some false beliefs on its run-time costs and practical viability. Our experimental results suggest that, depending on the underlying implementation, RUN may exhibit very modest kernel overhead, in some cases comparable to that of a partitioned algorithm, such as P-EDF. In addition, our experiments also confirmed that RUN may reduce the number of migrations in comparison to G-EDF. This latter observation is particularly relevant with a view to limiting the amount timing interference suffered by the system, as migrations are a potentially greater source of interference than preemptions.

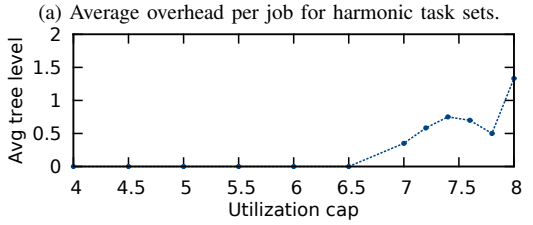
Our experiments also suggest that the number of preemptions and migrations (and schedule overhead) in RUN may depend not only on the way the off-line reduction tree is built (i.e., packing heuristic) but also on the task set characteristics and, in particular, on the distribution of task periods. Overly fragmented scheduling intervals as those induced by non-harmonic distributions imply more frequent updates to the reduction tree, which in turn cause a reschedule of the system.



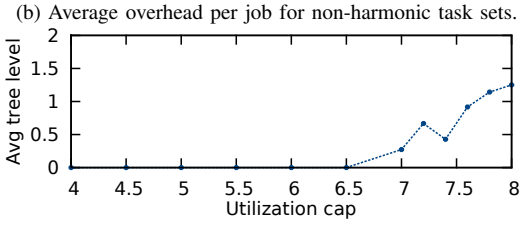
(a) Average overhead per job for harmonic task sets.



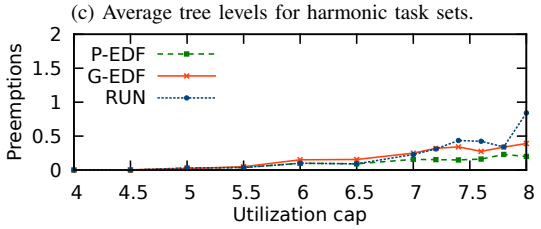
(b) Average overhead per job for non-harmonic task sets.



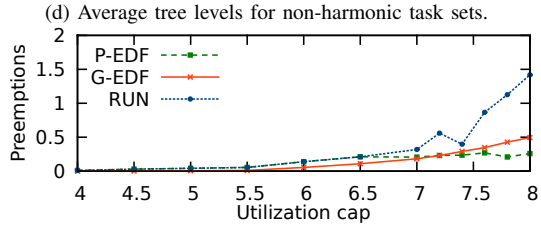
(c) Average tree levels for harmonic task sets.



(d) Average tree levels for non-harmonic task sets.



(e) Average preemptions per job for harmonic task sets.



(f) Average preemptions per job for non-harmonic task sets.

Fig. 8. Per-job scheduling overheads.

As future work, we want to evaluate RUN against U-EDF: bringing U-EDF on top of LITMUS$^{\mathrm{RT}}$ will be our first step in that direction. Moreover, we plan to deepen our understanding of how *good* reduction trees can be built that further reduce the number of preemptions and migrations.

Acknowledgment

The work presented in this paper has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA, www.proxima-project.eu).

References

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, 2011.

[2] J. Anderson, V. Bud, and U. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[3] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, 2009.

[4] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, 2007.

[5] P. Regnier *et al.*, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[6] J. Calandrino *et al.*, "LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers," in *IEEE 27th Real-Time Systems Symposium (RTSS)*, 2006.

[7] LITMUS$^{\mathrm{RT}}$, "The Linux Testbed for Multiprocessor Scheduling in Real-Time Systems," 2014, http://www.litmus-rt.org/.

[8] A. Bastoni, B. Brandenburg, and J. Anderson, "Is semi-partitioned scheduling practical?" in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[9] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Proceedings of the 9th International Parallel Processing Symposium*, 1995, pp. 280–288.

[10] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, pp. 600–625, 1996.

[11] J. Anderson and A. Srinivasan, "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks," in *ECRTS*, 2001.

[12] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *12th IEEE Internat. Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2006.

[13] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *IEEE 29th Real-Time Systems Symposium (RTSS)*, 2008, pp. 157–169.

[14] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers," in *IEEE 31st Real-Time Systems Symposium (RTSS)*, 2010.

[15] H. Chishiro, J. Anderson, and N. Yamasaki, "An evaluation of the RUN algorithm in LITMUS$^{\mathrm{RT}}$," in *WiP Session RTSS*, 2012.

[16] G. Nelissen *et al.*, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS*, 2012.

[17] P. Regnier, "Optimal multiprocessor real-time scheduling via reduction to uniprocessor," Ph.D. dissertation, UFBS, 2012.

[18] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[19] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[20] N. C. Audsley, I. J. Bate, and A. Burns, "Putting fixed priority scheduling theory into engineering practice for safety critical applications," in *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1996.

[21] L. C. Briand and D. M. Roy, *Meeting Deadlines in Hard Real-Time Systems*. IEEE Computer Society Press, 1997.

[22] S. Altmeyer and C. Burguiere, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *ECRTS*, 2009.