Università degli Studi di Padova
Dipartimento di Matematica

Corso di Laurea Magistrale in Informatica

RUN (Reduction to UNiprocessor): implementation and evaluation

Candidato:
Davide Compagnin
Matricola 623226

Relatore:
Tullio Vardanega

Anno Accademico 2012–2013

Contents

Abstract

The Reduction to UNiprocessor (RUN) scheduling algorithm represents an original approach to multiprocessor scheduling that exhibits the prerogatives of both global and partitioned algorithms, without incurring the respective disadvantages. As an interesting trait, RUN promises to reduce the amount of timing interference between cores, while being at the same time theoretically optimal. However, RUN has also raised some doubts on its practical viability in term of incurred overhead and required kernel support. Surprisingly, to the best of our knowledge, no practical implementation and empirical evaluation of the algorithm have been presented yet. In this thesis we present the first solid implementation of RUN and extensively evaluate its performance against P-EDF and G-EDF, with respect to empirical 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 moderate overheads and interference and supporting much higher schedulable utilization than its partitioned and global counterparts.

Acknowledgements

I would like to express the deepest appreciation to my advisor Prof. Tullio Vardanega for the continuous effort in supporting my work, his motivation, enthusiasm and immense knowledge. Without his guidance and persistent help the results produced in this thesis would not have been possible.

I also wish to thank Enrico Mezzetti as assistant supervisor, without his support and technical experience I would not have been able to finish the numerous experiments conducted and to submit the paper at the conference.

My most heartfelt thanks, however, go to my girlfriend Irene that has always been close to me, as well as, she helped me to correct the english language in this thesis.

A special gratitude I give to my parents for their endless love and their financial support during all these years.

A final thanks I give to all my friends, colleges and especially to the members of the jazz band to which I belong.

Padua, October 2013

Chapter 1

Introduction

Compelling energy, performance and availability considerations push towards the migration from relatively simple single processors systems to more complex multiprocessors platforms. The latter in fact provide the necessary computational power without incurring thermal and energy drawbacks. Research on real-time scheduling has been mainly devoted to uniprocessor systems, and today reached a reasonably maturity, with a large number of key results. Only after the advent of multicore systems, called *multicore revolution* in [Bertogna, 2008], research in the multiprocessor scheduling problem has regained greatly interest. Despite the great research efforts and recent advances and results, scheduling and schedulability analysis of multiprocessor systems have not reached the same impact level as that of single processors [Davis and Burns, 2011]. In fact, the deployment of hard and soft real-time systems on top of multiprocessor systems requires adequate scheduling policies and new analysis techniques to guarantee that their real-time requirements can be always met.

1.1 A background on real-time scheduling

A real-time system, as defined by Burns and Welling [Burns and Wellings, 2009], is

"*an information processing system which has to respond to externally generated input stimuli within a finite and specified period. The correctness depends not only on the logical result but also on the time it was delivered. The failure to respond is as bad as the wrong response*".

Today, many different applications which use real-time systems are found in automotive electronics, avionics, telecommunications, space systems, medical imaging and consumer electronics.

In order to guarantee time requirements of real-time systems and to reason about its properties, a system model has to be defined. This section briefly describes the reference real-time system model [Liu, 2000] besides basic terminology and notation. Afterwards, it presents an overview of the multiprocessor scheduling algorithm, focusing on the semi-partitioning approach.

According to [Liu, 2000], a real time system is characterized by three elements:

1. a *workload model* that describes the application supported by the system
2. a *resource model* that describes the system resources available to the applications
3. algorithms that define how the application system uses the resources all times

The *workload model* is based on the notion of *job* and *task*.

The *job*, denoted J, is a finite sequence of instructions, conceptually an unit of work, that is scheduled and executed by the system. A job is, for example, the computation of a control-law, the transmission of a data packet, the retrieval of a file, and so on.

Job is characterized by three values, as depicted in Figure 1.1: a release time $J.r$, a worst-case execution time $J.c$ and an absolute deadline $J.d$. The *release time* is the instant of time when the job becomes available for execution, while the *absolute deadline* is the instant of time by which its execution is required to be completed. The worst-case execution time is a safe over estimation of the maximum amount of time required by a job to complete its execution.

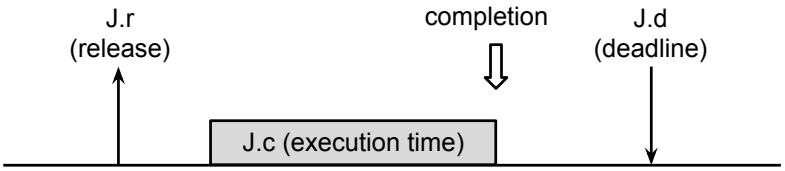


Figure 1.1: Job model.

Unpredictability of the execution time is due to two main reasons: the hardware and the job semantic. Hardware enhancing features, such as the cache unit, may have a strong impact on the time of execution of a job and, since it has a state that depends on the execution history, it makes the system behaviour non-deterministic. The second problem, instead, emerges in jobs containing branch instructions. The execution flow may change following different paths that may depend on the input

data, potentially causing to have very different execution time. Therefore, the worst-case execution time (WCET) has to be taken to make the system model deterministic.

Execution of jobs can be interleaved. The scheduler may suspend the execution of a less urgent job in order to execute a more urgent job. Once the more urgent job completes, the less urgent job can resume execution. This interruption of job execution is called *preemption*. A job is *preemptable* if its execution can be suspended at any time. On the contrary, a *nonpreemptable* job executes from start to completion without any interruptions. In a system of two or more processors, a job may resume execution in a different processor. In this case, it is said that the job *migrates* from a processor to another.

The *task*, denoted τ, is a functional and architectural unit of composition. It *releases* jobs to support a function of the system. A system, not surprisingly, is called *task set*, denoted $\mathcal{T}$. Tasks are classified according to the time behaviour of the job release:

- tasks that release jobs at regular time intervals are called *periodic*. They are characterized by a *period* T and a worst-case execution time C
- tasks that release jobs ensuring a minimum time interval are called *sporadic*. They are characterized by a *minimum inter-arrival time* T and a worst-case execution time C
- tasks without any constraints are called *aperiodic* and are usually used to model non real-time workload

A real-time workload modelled using periodic tasks with implicit deadline is well known as *periodic task model*. This is the model considered in this dissertation.

A deadline value D_i may also be specified for all the above tasks and determines the actual deadline for the released job by relation $J.d = J.r + D_i$. The job deadline is then *absolute*, while the task deadline is *relative* to the release instant of the job. Deadline can be *constrained* if it is smaller than period ($D_i < T_i$), *implicit* if it is equal to the period ($D_i = T_i$) or *arbitrary* if it is bigger than period ($D_i > T_i$). If not specified, deadline is assumed to be implicit.

The *utilization* or rate of a task τ_i, denoted $\rho(\tau_i)$ is the ratio between the execution time and period: $\rho(\tau_i) = \frac{C_i}{T_i}$.

A system is classified according to its timing constraints. A system is said to be:

- *hard* real-time, if the failure to meet a deadline is considered to be a fatal fault
- *soft* real-time, if missing a deadline is undesirable
- *firm*, if missing a deadline does not cause a fatal fault but the usefulness of the result drop immediately to zero

The *resource model* categorizes the system resources in *active* and *passive* resources. Active resources are able to execute jobs, or alternatively, a job must acquire an active resource to make progress toward completion. For this reason, they are commonly called *processors* or processing elements. Passive resources may be required from an executing job to deliver its function and typically may be reused if use does not exhaust them. Passive resources are, for example, memory, mutexes and lock.

The number of processors and their configuration are commonly used to characterize the computing platform. A computing platform composed by only one processor is named *uniprocessor* platform, instead, if it is composed by two or more processors, it is named *multiprocessor* platform. A multiprocessor platform can be *identical*, *uniform* or *heterogeneous*, depending on the processor configuration.

- A multiprocessor platform is said to be identical or homogeneous if the worst-case execution time of a task does not depend on the particular processor on which it is executed. All the processors exhibit the same processing capabilities (i.e., cache size and hierarchy, I/O buses, instructions set, etc.) and speed, so they are interchangeable. All tasks can be executed on all processors without any restriction.

- A multiprocessor platform is said to be uniform if the worst-case execution time vary in function of the processor on which they are running. Processor exhibit the same processing capabilities but different speeds. All tasks can be executed on all processors without any restriction.

- Finally, a multiprocessor platform is said to be heterogeneous or unrelated if processors may have different capabilities and different speed. Some tasks may therefore not be able to execute on some processors in the platform. To this categories belong for example platform that combine several CPUs and GPUs resources.

Due to its simplicity, the majority of works in literature refer to the identical multiprocessor platform, which is also assumed in this thesis.

This type of platform is also known as *symmetric multiprocessor platform*. An SMP is a tightly-coupled multiprocessor system where multiple processing units are connected at the bus level and have a shared central memory that is uniformly accessed. A *multicore* platform is a slightly variation of the SMP that concerns the access to the cache. In the SMP, each CPU accesses its private cache, while in the multicore, the cache is shared.

The *scheduler* is the part of the system that takes scheduling decisions according to a scheduling algorithm. Intuitively, a *real-time scheduling algorithm* determines

which task has to be executed on each processor at any time, ensuring that every job can successfully complete the execution before its deadline. At a given time a job is *scheduled* if it is assigned to a processor, and the total amount of processor time assigned to a job must be equal to the job execution time.

There are many ways to formally define a scheduler. According to [Regnier *et al.*, 2011a], a scheduler is a composition of a *scheduling function* that determines the scheduled jobs at a given time t, and a *assigning function* that allocates the scheduled jobs to a processor. The scheduling function, denoted Σ, is a function from all non-negative times t to the power set of the set of jobs $\mathcal{J}$, such that $\Sigma(t)$ is a subset of jobs in $\mathcal{J}$. The assigning function, denoted Δ, assigns a job scheduled at time t to a processor π_i. An assignment of jobs to processors resulting from the applications of these functions is called *schedule*. The schedule has to satisfy the following conditions in order to be *valid*:

- every processor is assigned to at most one job at any time
- every job is assigned at most one processor at any time
- no job is scheduled before its release time
- the total amount of processor time assigned to every job is at most its worst case execution time
- a job can only execute on a single processor at any time

A scheduling algorithm is correct as long as it produces a valid schedule. We implicitly assume this property to all the algorithms addressed in this thesis. A valid schedule is also *feasible* if every job completes by its deadlines, that is all the time constraints are met. A task set is said to be *schedulable* according to a scheduling algorithm if, when applied, it produces a feasible schedule. It is worth noting that, unlike schedulability, the feasibility is an intrinsic property of the task set and does not depend on the scheduling algorithm.

Scheduling algorithms are commonly compared by their ability to find a feasible schedule for a given task set, whenever such schedule exists. On the one hand, algorithms that always produce a feasible schedule for a given feasible task set are said *optimal*. As stated in [Horn, 1974], the necessary and sufficient condition for the feasibility of implicit-deadline periodic task sets $\mathcal{T} \ni \{\tau_i\}_{i\in\{1,n\}}$ is:

$$\sum_{i\in\{1,n\}} \rho(\tau_i) \leq m$$

where m is the number of processors. On the other hand, a not optimal algorithm can make use of an offline schedulability test in order to determine if a feasible

schedule is produced for a given task set.

Finally, a scheduling algorithm is said to be *work-conserving* if it leaves one or more processors idle when tasks are ready to execute.

After giving these basic concepts, we focused on the problem of scheduling a set of tasks assuming:

- A platform comprised of $m > 1$ identical processors
- A preemptive and independent job model with no migration restriction
- A periodic task model with implicit deadlines
- Online scheduling algorithms

1.2 Real-time scheduling on multiprocessor

Research on multiprocessor scheduling has been primarily devoted to the main classes of *partitioned* and *global* scheduling algorithms. Neither partitioned nor global approaches, however, show decisive advantages over the other [Davis and Burns, 2011]. Intrinsic limitations of both classes of algorithms have been widely acknowledged [Bastoni *et al.*, 2011; Davis and Burns, 2011].

Partitioned approaches rely on a static mapping that assigns tasks to processors, so that each task can be only scheduled on the processor it has been assigned to, and task migration is prohibited. In other words, they reduce the multiprocessor scheduling problem to canonical uniprocessor scheduling ones by solving an off-line task allocation, which is substantially equivalent to the NP-hard bin-packing problem. Hence, a wealth of real-time scheduling techniques and analyses for uniprocessor systems can be applied to schedule each processor separately. From a practical perspective, a per-processor ready queue is used, simplifying deployment and reducing the overhead of manipulating a single global queue. The main disadvantages of partitioned approaches (e.g.,, P-EDF) are:

- they must undergo a complex and inherently iterative task allocation phase whose solution cannot guarantee total utilization comparable to those obtainable with global algorithms. As showed in [Andersson *et al.*, 2001], the worst-case utilisation bound for a periodic task set with implicit deadlines is:

$$\sum_{i \in \{1,n\}} \rho(\tau_i) = \frac{(m+1)}{2}$$

- demanding the task allocation problem to the system designer, as a system-level heuristic design, is not a sustainable practice within a standard development process

- allowing tasks to share global resources, which have to be accessed in mutual exclusion, produces drastically impacts on the schedulability. In fact, multiprocessor variants of Priority Ceiling Protocol suffer of several different components of blocking as summarized in [Gai *et al.*, 2003], which produce very pessimistic assumptions and complicate the schedulability analysis.

Global algorithms, instead, do not pose any fixed task-to-processor allocation, and tasks or jobs are allowed to *migrate* from one processor to another, upon task preemptions and suspensions. While being naturally work-conserving and thus able to guarantee higher utilization, they bring in the additional overhead incurred by system-wide scheduling data structures as well as the potentially destructive effects of the migration. Global algorithms, such as 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 have been proved for the family of *proportional fairness* algorithms, i.e., PFair [Baruah *et al.*, 1996; Baruah *et al.*, 1995] and its derivatives (e.g., [Anderson and Srinivasan, 2001]). PFair algorithms approximate the theoretical *fluid scheduling* model, according to which all tasks execute in any time interval at the steady rate proportional to their utilization. The time line is divided into equal length quanta or slots. At each time quanta, the scheduler allocates tasks to processors potentially causing several migrations. DP-Fair algorithms relax the time-continuity constraint of the fluid model and prove that optimality could be reached by simply ensuring the fairness at each shared deadline, called *system deadline*, of jobs released in the system. Unlike PFair, time slices are bounded by two consecutive system deadlines and execution time in each time slice is determined so as to ensure that all deadlines will be met. In practice, both PFair and DP-Fair approaches waste majority of the theoretical utilization bound due to the need of synchronization on all time-slot boundaries, however small, and to the ensuing migration overhead.

The fact that neither partitioned nor global scheduling offer a satisfactory and general solution to the multiprocessor scheduling problem comes out in favour of 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-WM [Kato *et al.*, 2009b] or NPS-F [Bletsas and Andersson, 2009] where only a small subset of tasks is allowed to migrate, thereby improving the schedulability, and *clustered* variants of global algorithms, such as

C-EDF [Calandrino *et al.*, 2007], where tasks are only allowed to be scheduled on (and thus migrate within) a subset of the actual processors. Although in literature are presented several semi-partitioned approaches, at the best of our knowledge, there is not a comprehensive taxonomy and categorization. Commonly, in semi-partitioned approaches, tasks whose allocation is statically defined are called *fixed*, in contrast to *migratory* tasks that are free to migrate.

The EDF with Window-constraint Migration (EDF-WM) is designed to support the hard real-time scheduling of sporadic tasks. It basically assigns each task to an individual processor according to a bin-packing heuristic, as long as it can be done. A task becomes migratory when no individual processor has enough remaining capacity to accept the full share of the task. Figure 1.2 shows an example of the approach

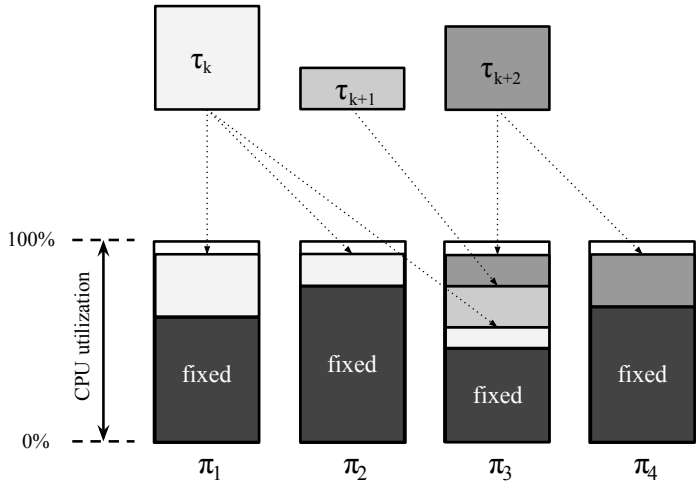


Figure 1.2: EDF-WM task allocation.

on four processors. This example assumes that first $k-1$ tasks are already assigned (fixed) to the processors. Then, it is considered a case in which the task τ_k cannot be assigned to any individual processors. Traditional partitioning would have failed, on the contrary EDF-WM splits τ_k and assigns it to processors using a first-fit heuristic, for instance π_1, π_2 and π_3. In the same way, τ_{k+1} and τ_{k+2} are split and assigned to the remaining π_3 and π_4 processors. It is important to note that a processor may include more than one migratory task, and such a migratory task is not necessarily split across continuous processors. The split approach effectively creates a sequence of sub-tasks that are assigned to distinct processors. In order to avoid the parallel execution, the algorithm splits the deadline of each migratory task into the same number of processors among the task which has been assigned to. The job scheduling window d_k is then sliced according to these "virtual deadlines", as depicted in

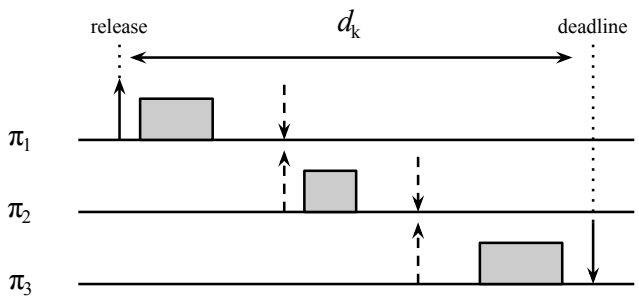


Figure 1.3: EDF-WM scheduling window splitting.

Figure 1.3, and the EDF algorithm is applied to each local processor guaranteeing that multiple processors never execute the same task simultaneously. In [Kato *et al.*, 2009a] is proven that this approach dominates the partitioned scheduling approach.

The Notional Processor Scheduling - Fractional capacity (NPS-F) algorithm is another example of semi-partitioned approach designed to support the hard real-time scheduling of sporadic tasks. It is an evolution of the EKG algorithm [Andersson and Bletsas, 2008] and it achieves a higher utilization bound (potentially can be optimal) reducing preemptions. The task assignment phase is divided into two steps. The first step partitions the task set using a bin-packing heuristic, but unlike EDF-WM, tasks are assigned to virtual processors, called *servers*, of unitary capacity. Since no task splitting is performed at this step, at most $m' \in \{1, n\}$ servers are produced, where n is the size of task set. The second step increases the capacity of these servers over-provisioning the system in order to ensure schedulability (i.e., take the scheduling overhead into account). This operation is performed by an *inflation* function $I(\delta, c_i)$ where δ is a parameter that allows to increase or reduce the utilization bound of the algorithm causing respectively an increment or a reduction of preemptions and c_i is the capacity of the server. The m' inflated servers, called *notional processors*, are sequentially mapped onto the m physical processors, one at a time. Once the condition $\Sigma_{i=1}^{m'} I(\delta, c_i) \leq m$ is verified, the processor mapping is feasible. During execution, each server is scheduled in a time slot which depends on the δ parameter as well as a minimum system period. Once a server executes, it selects the packed ready task with the earliest deadline.

In this section, we have presented two semi-partitioned scheduling approaches: the EDF-WM and the NPS-F. Although these algorithms do not cover all the semi-partitioned approaches presented in the literature, they are sufficiently to observe some common characteristics:

- semi-partitioning are inherently more sophisticated than partitioned and global approaches. They perform both local and global scheduling in order to move tasks among the processors
- tasks are statically allocated to processors, and eventually split. The allocation is based on bin-packing heuristics
- the allocation precisely determines which tasks are migratory and the migrations across the processor.
- the online schedule is executed according to the offline task allocation.

Within this line of thought, the *Reduction to UNiprocessor* (RUN) algorithm, first introduced in [Regnier *et al.*, 2011a], represents an original and interesting approach. Although the author classifies RUN as a semi-partitioned algorithm we believe that it behaves very differently to the other semi-partitioned approaches. Basically, when applied to homogeneous multiprocessor systems, comprising m processors, RUN employs particular scheduling abstractions such as *primal* and *dual servers* to build an off-line data structure (*reduction tree*), which is used to reduce the scheduling problem to m single processor schedules and to guide scheduling decisions at run time. Those scheduling decisions are taken so as to minimize the number of migrations and thus the incurred overhead. Moreover, RUN is claimed to be an *optimal* algorithm for multiprocessor scheduling, that is it is always able to produce a valid schedule whenever a task set is feasible.

1.3 The challenge taken in this work

The characteristics of RUN make it an extremely promising trade-off between global and partitioned algorithms. However, despite having been widely appreciated for the elegance of the proposed solution, RUN has also given rise to some doubts on its practical implementation, mainly due to the building process and data representation of the reduction tree, and the overhead expected from frequently updating the latter. In fact, the real-world overhead in the implementation of multiprocessor algorithms plays an essential role, as pointed out in [Bastoni *et al.*, 2011]; in particular, the latter study presents a systematic description of typical overhead-related issues suffered from a number of different algorithms.

Those doubts can only be confirmed or refuted by an empirical evaluation of the algorithm. Quite surprisingly, to the best of our knowledge, neither an empirical evaluation of RUN or a comparison with other algorithms has been performed yet.

In our implementation effort we use LITMUS$^{\mathrm{RT}}$ [Calandrino *et al.*, 2006; LITMUS$^{\mathrm{RT}}$, 2013], a real-time extension to the Linux kernel published by UNC, which exploits

a plug-in mechanism for the definition of multiprocessor scheduling algorithms.

There are four main reasons to justify this choice:

- currently, at the Department of Mathematics in Padua, the known real-time kernels do not yet provide a mature support to the multiprocessor scheduling, especially for general purpose architectures; LITMUS$^{\mathrm{RT}}$ on the contrary, takes advantage of the great support of the Linux kernel to many different multicore architectures. Moreover, it relaxes the intrinsic partitioned nature of Linux, providing a powerful migration support for global and semi-partitioned algorithms
- it allows to easily implement and deploy scheduling algorithms hiding many of the difficulties related to the low-level programming
- it is very effective and it enables the implementation of many scheduling algorithms, global and partitioned, without introducing particular restrictions
- it provided a rich set of tracing tools that simplify the offline analysis and comparison of scheduling algorithms

Before entering into the implementation details of the Chapter 2, we first provide a short description of RUN, in its original formulation, and an overview of the LITMUS$^{\mathrm{RT}}$ environment.

1.3.1 The RUN algorithm

The underlying principle of RUN is, not surprisingly, the *reduction*. As the name suggests, reduction allows to solve a multiprocessor scheduling problem by reducing it to an equivalent uniprocessor problem. From a theoretical point of view, this definitively proves that the multiprocessor scheduling problem is not more difficult than the uniprocessor one.

The reduction is preformed off-line upon the task set, similarly to partitioning, by exploiting the *dual scheduling* principle, introduced in [Levin *et al.*, 2009], in a more original way, and by introducing an original task model called *fixed-rate* that is a generalization of the periodic task model.

The reduction produces a data structure called *reduction tree* which is essential to make on-line scheduling decisions. Since the reduction is not univocal and acts on the task interference, it may change the efficiency of the algorithm during the on-line phase in terms of preemptions ad migrations. Although an in-depth knowledge of the relation between the off-line and the on-line phase makes RUN very attractive, this thesis is focused on providing a baseline RUN implementation.

Unlike DP-Fair, fairness is not ensured at every system deadline, however, it achieves optimality by a weak notion of fairness which is encapsulated in the task model. Reduction on the one hand, and weak fairness on the other hand, significantly reduce preemptions and migrations.

Although RUN assumes the real-time system model described on Section 1.1, it introduces a new task model called *fixed-rate*. A *fixed-rate task* is a task whose execution requirement is specified in terms of the processor utilization within a given interval. Then, it is completely characterized by two values: an utilization $\rho \leq 1$ and an unbounded set D of positive real numbers termed deadlines. A fixed-rate task $\tau(\rho, D)$ can only release a job in correspondence of a deadline and has to complete before the next deadline. Two consecutive deadlines D_{i-1} and D_i bound the scheduling interval, also called *scheduling window*, for a job J which execution time $J.c$ equals to $\rho(J.d - J.r)$ where $J.r = D_{i-1}$ and $J.d = D_i$. The fixed-rate task model is clearly a generalization of the periodic task model with implicit deadlines, even though, the definition given in [Regnier *et al.*, 2011a] ambiguously refers to *non-periodic* tasks. However, the conditions under which a sporadic task can be completely represented by a fixed-rate are not yet clear.

The concept of fixed-rate task is fundamental in the formulation of the algorithm because it allows to easily combine execution requirements of several tasks grouped together into a *server*. The *server* is a scheduling abstraction which acts as a proxy for a collection of client tasks and behaves like a fixed-rate. Given a set of fixed-rate tasks $\mathcal{T}$ with utilization $U \leq 1$, a server is a fixed-rate task with utilization U and deadlines equal to the union of the client's deadlines. In order to schedule such aggregation of tasks, the server is equipped with a uniprocessor scheduling policy (i.e., EDF) since its utilization is at most 1. A server is said *unit-sever* if its utilization equals to 1.

Offline concerns

As introduced, the reduction is based on the *dual scheduling equivalence* which sates that solving the problem of scheduling a given task set $\mathcal{T} \ni \{\tau_i\}_{i\in\{1,n\}}$ on m processors is equivalent to that of scheduling the *dual* task set $\mathcal{T}^* \ni \{\tau_i^*\}_{i\in\{1,n\}}$ on $n - m$ processor. The *dual* τ_i^* of the task τ_i is a task characterized by the same set of deadlines but a complementary utilization. Let $\rho(\tau_i)$ denote the utilization of τ_i, then $\rho(\tau_i^*) = 1 - \mu(\tau_i)$. τ_i^* represents exactly the τ_i's idle time. Therefore, every time the task τ_i^* is running in the dual schedule then the task τ_i must stay idle in the actual schedule (also called *primal* schedule). On the contrary, if τ_i^* is idle in the dual schedule then τ_i must execute. Consequently, the primal schedule can be derived from the dual schedule.

It is worth noting that a dual task set is feasible in $n - m$ processor. In fact, as proven in [Regnier *et al.*, 2011a], the total utilization of the dual task set, denoted U^*, is equal to $n-U$, where U is the total utilization of the primal task set. Consequently, in a fully utilized system $U = m$, the dual task set is feasible in $n - m$ processors. On this basis, every time that $n - U < m$, the problem has been reduced.

In order to guarantee this relation, the number of task n has to be kept sufficiently small, therefore, RUN *packs* tasks into scheduling abstractions called *servers* as described before. In other words, it applies a bin-packing algorithm to a set of many light tasks obtaining an equivalent task set of few heavy tasks. This operation is called PACK in [Regnier *et al.*, 2011a] while the operation that produces the dual task set is called DUAL. By alternately packing tasks into severs (PACK) and reformulating the scheduling problem into its dual (DUAL), the number of processors is progressively reduced and eventually reaches one [Regnier *et al.*, 2011a], thus reducing the initial problem to a uniprocessor scheduling problem.

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, the so-called *reduction tree*, which is exactly the data structure that is used to derive scheduling decision at run time, is obtained.

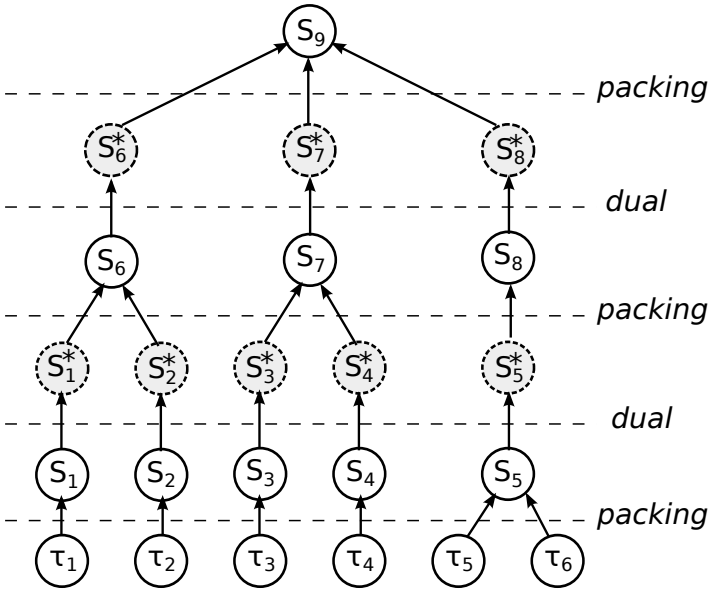


Figure 1.4: RUN reduction.

Figure 1.4 shows an example of reduction and the resulting reduction tree. Reduction proceeds in a bottom-up fashion. As a first step, tasks are packed into level-0 servers under the condition that the cumulative utilization cannot exceed 1: for example τ_1 and τ_2 are packed together into S_1 (and so do τ_5 and τ_5 in S_4), while 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. The next step consists in switching to the dual representation of the problem, which originates the respective set of dual servers S_i^*. This sequence of steps is then repeated until we get a set of servers S_5^*, S_6^*, S_7^* with a total utilization 1, thus feasible on a single processor. By packing the last level of dual server into a *unit* server S_8 we set down the root of the reduction tree, whose leaves regroup the original task set. The assumption on the full system utilization guarantees the termination of the construction of the reduction tree.

We believe that all the RUN principle synthesized by the reduction tree determines a partition of the system. Formally, let $\mathcal{T}$ be a task set of size n whose utilization equals the number of processors m. A reduction of $\mathcal{T}$ produces a partition Γ of $\mathcal{T}$ such that $\Gamma = \{\Gamma_1, \ldots, \Gamma_k\}$ and for each $i \in [1, k]$, $\rho(\Gamma_i) \in \mathbb{N}$. Each Γ_i reduces to uniprocessor therefore is feasible on $m_i < m$ processors where $m_i = \rho(\Gamma_i)$. Consequently, a subset of system processors, also called *cluster*, is associable to each Γ_i, guaranteeing that a cluster is exclusively assigned to a Γ_i.

This kind of partitioning is a very important result: tasks are free to migrate only *inside* their processor cluster, but tasks in a partition are isolated respect to the other clusters. Thus, RUN has been classified under the umbrella of semi-partitioned algorithms. Nonetheless it is clear that RUN is not a semi-partitioned approach in the common acceptation and falls into a separate category.

Online concerns

At run time, scheduling is performed according to the information in the reduction tree, which is updated during the execution. 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 which selected for execution ($circled$ in RUN speech) 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 at each instant t. The correctness of the relationship between primal and dual schedule, is guaranteed by RUN assumptions on full utilization and constant execution requirements.

Figure 1.5 shows a possible task selection as determined by the state at time t of the same reduction tree reported in Figure 1.4. In this case, tasks τ_3 and τ_6 have been selected for execution.

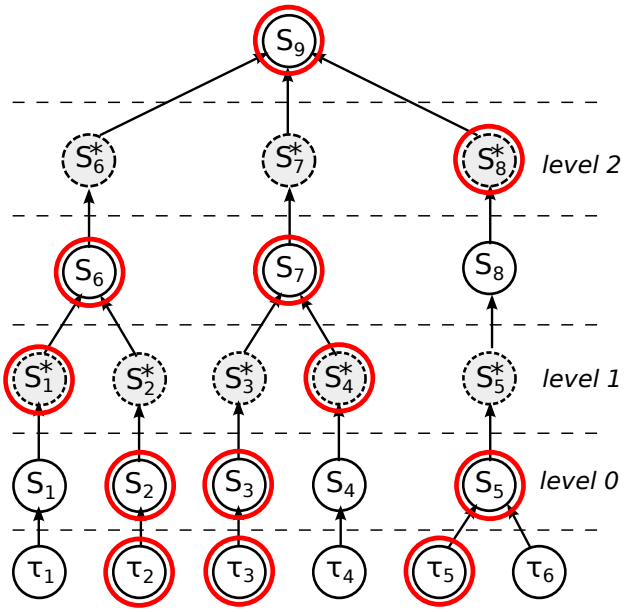


Figure 1.5: RUN scheduling.

In contrast with G-EDF, RUN is prevented from taking greedy scheduling decision by the proportional 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 according to I.

Considerations on the reduction and task allocation

The RUN off-line phase of aims to:

- produce a reduction to *unit-processor* reducing a problem of multiprocessor scheduling into m uniprocessor problem. Each of these can be locally schedule on a processor;
- aggregate tasks into servers with a unitary load (*unit* server) to ensure, during the on-line phase, the *exactly* processor demand for each job;
- express the relationship among tasks and guides their allocation to avoid to make greedy scheduling choices

Therefore, jobs can be allocated online without exceeding the capacity assigned to them and without missing their deadline, achieving the optimality. The dual task concept guarantees this behaviour since it prevents that on one *unit-processor* (that

acts as container of unitary capacity) can be allocated a workload (content) that exceeds its capacity.In this way, a *unit-processor* determines a subset of the task set which load never exceeds the capacity of the container and consequently it can be filled at run-time according to rules of local scheduling. The job migration is then possible, since the content of each processor is decided at run-time , but still remains coherent to the off-line phase. This distinguishes RUN from other semi-partitioned algorithms in which most of the task are statically allocated.

As observed in [Regnier *et al.*, 2011a], the efficiency of the algorithm, in terms of preemptions and migrations, depends on the reduction tree. The temporal and spatial isolation among tasks, e.g. of the cache, varies too. More a task is isolated less interference it suffers from the events generated by other tasks.

Considerations on the design

RUN is a hybrid approach that may behave more or less similarly to a global o partitioned algorithm: the packing step which seems to group RUN with partitioned approaches is actually quite particular as tasks are not associated to processors but to servers, which in principle are not pinned to any particular processor.

On the one hand RUN is global in the sense that scheduling decisions are taken globally and the packing is much less critical to the final performance of the algorithm as compared to, for example, P-EDF: it may influence the shape of the reduction tree, but it cannot compromise the validity of a schedule (when P-EDF packing does). On the other hand, with favourable tasks set, the reduction tree may produce exactly the same behaviour as P-EDF. Being able to be partitioned or global depending on the task set is a very valuable property of this algorithm. As observed in [Regnier, 2012] what is actually partitioned is the concept of proportionate fairness which is proportionate to servers (i.e., group of tasks). These original features in RUN were also the main source of implementation challenges, as we will discuss after a brief introduction to the LITMUS$^{\mathrm{RT}}$ environment in Section 1.3.2.

1.3.2 The LITMUS$^{\mathrm{RT}}$ environment

The **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems is an extension to the Linux kernel [Torvalds, 1994] developed by University of North Carolina at Chapel Hill for the development and evaluation of scheduling algorithms and locking protocols in multiprocessor systems. LITMUS$^{\mathrm{RT}}$ provides, among others, a set of components (e.g., queues, timers, etc.), an augmented set of system calls for real-time tasks and a simple plugin interface which can be exploited to activate at run-time a specific scheduling policy. These features enable the definition of a large class of user-defined scheduling policies. Several scheduling algorithm

have been in fact already developed and made available to the public along with LITMUS$^{\mathrm{RT}}$ releases [LITMUS$^{\mathrm{RT}}$, 2013]. Implemented algorithms include global, partitioned and clustered variants of EDF as well as an implementation of the partitioned fixed-priority (P-FP) and P-Fair algorithms. Moreover, LITMUS$^{\mathrm{RT}}$ offers a score of interesting tracing and debugging capabilities that form an extensively automated framework for both the development and evaluation of scheduling primitives and kernel overhead.

The Linux scheduling framework

Due to unpredictable issues, such as the non-preemptability of critical sections and the unpredictability of the interrupt management duration, the Linux kernel is not suitable for strictly hard-real time applications. Despite these limitations, it can can handle a large and important subset of real-time applications, as noted by [McKenney, 2005]. Moreover, its strength is that it can scale without particular restrictions on multicore platform.

The Linux scheduler is organized as a hierarchy of *scheduling classes*. Each scheduling class defines a scheduling policy and also provides a series of functionalities: adding a task to be scheduled, pulling the next task to be run, yielding to the scheduler, and so on. Each scheduler class is linked to another in a single linked list. Processes can be scheduled according to different scheduling classes, but they must belong to a single scheduling class at a time. Whenever a scheduling decision is required, classes are traversed from the top to the bottom of the hierarchy until a pending process is found. Therefore, processes at lower-priority scheduling classes are only considered if higher-priority scheduling classes are idle.

From an implementation point of view, the Linux scheduler is fundamentally partitioned, favouring local process executions. A *runqueue* is associated with each processor and contains the state of each scheduling class, which, among other, includes a processor-local ready queue, the current time, scheduling statistics. Moreover, the runqueue is protected against concurrent access by a lock. Since a processor can belong to only one processor at the same time, when a migration is required two runqueue locks must be acquired. As a consequence, global scheduling incurs in some difficulties related to the design of the Linux kernel.

It is important to note that theoretical model and schedulability tests do not assume scheduling overhead. In practice, scheduling events are not atomic and there are several situations during which the scheduler can be in a transitory state. For instance, the notion of when a job is scheduled is not uniquely defined, because a task can be scheduled but currently suspended, or selected by the scheduler but

not yet dispatched. This non-atomicity of scheduling events considerably complicates scheduling decisions because it is possible to take wrong scheduling decisions because the system is in an inconsistent state.

Architectural overview

The LITMUS^{RT} infrastructure consists of four main parts as depicted in Figure 1.6:

- a core infrastructure
- a number of scheduler plugins
- a user-space interface
- a user-space library and tools

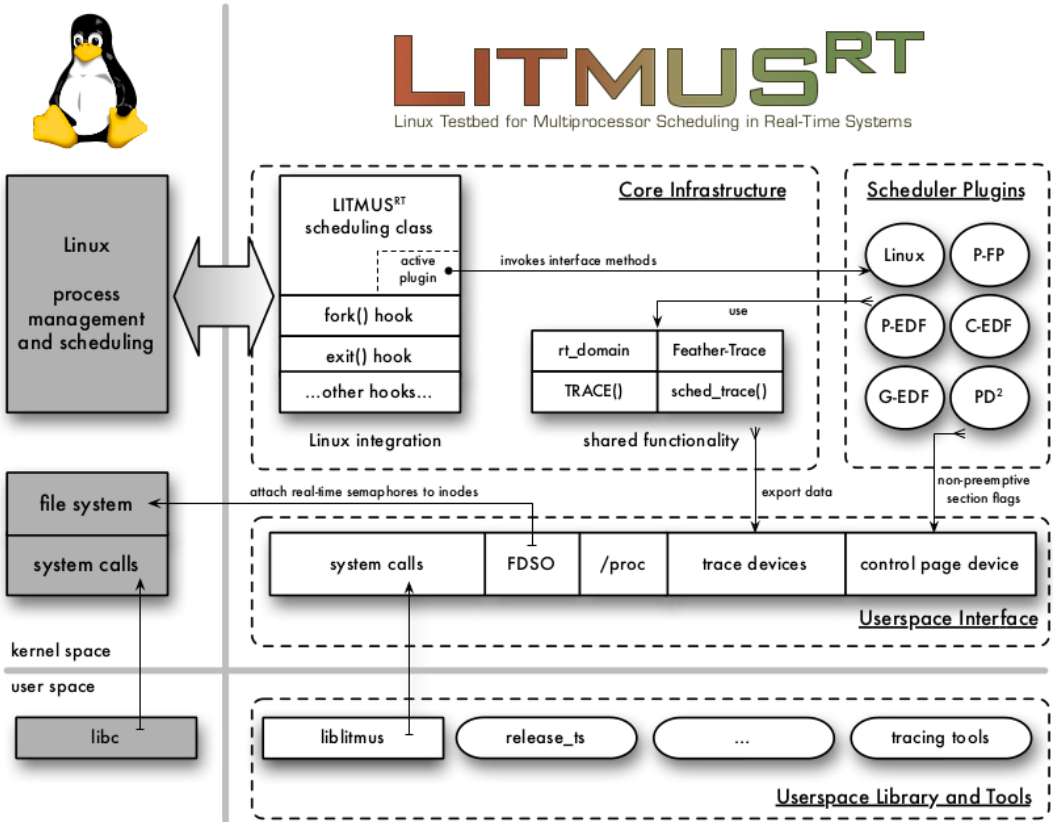


Figure 1.6: The LITMUS^{RT} infrastructures.

This section describes the core infrastructure and the real-time domain abstraction from an high point of view, whereas technical details, such as the plugin interface and the user-space interface, can be found in Appendix A.

The core infrastructure aims to simplify plugin development and maintenance as well as correctness **(i)** by hiding most of the complexities related to the Linux scheduling framework and **(ii)** by providing very common reusable data structures and mechanisms facilitating the reuse of code. It is important to note that since LITMUS$^{\mathrm{RT}}$ exploits the Linux design, real-time tasks are implemented as standard Linux processes: the process *main* executes an infinite loop where each iteration corresponds to release a new job. For this reason, we say process meaning real-time tasks when the focus is on the implementation.

Regarding (i), it adds a new scheduling class on top of the Linux hierarchy overriding default scheduling policies, and consequently, lowering the priority of Linux processes, which can only execute when no LITMUS$^{\mathrm{RT}}$ real-time processes are running. Thus, a system call allows to promote Linux processes from the *background mode* (executing as best-effort workload) to the *real-time mode* and vice-versa. On the contrary, in the absence of real-time processes, system behaves like a normal Linux system greatly simplifying system administration and the preparation and evaluation of experiments. However, LITMUS$^{\mathrm{RT}}$ scheduling class does not implement any particular scheduling policy. It implements an abstract common plugin-based interface of scheduling that enables to embed scheduling logic on a single component (the plugin). In other words, it is able to defer all scheduling decisions to the active plugin. This plugin interface abstraction is precisely described below. There are numerous benefits on introducing LITMUS$^{\mathrm{RT}}$ as an intermediate layer. It protects against version changes of the underlying Linux kernel. In addition, it provides a solid migration support for global and semi-partitioned algorithms, as well as a runtime switching mechanism for the scheduling plugins.

Regarding (ii), two interesting parts are the plugin interface specification and the real-time domain abstraction which provide reusable ready and release queues, explained in the following section.

The real-time domain abstraction

Each scheduler plugin tipically requires a mechanism both to order ready jobs and to queue jobs for future time-based releases. Not surprisingly, LITMUS$^{\mathrm{RT}}$ implements two very common data structures: the ready and the release queue. When a job is released, it must be transferred from the release queue to the corresponding ready queue, and the scheduler needs to be invoked to check if a preemption is required. All these mechanism are abstracted by a reusable component called real-time domain (`rt_domain` in the code).

The `rt_domain` implementation uses the Linux *hrtimers* subsystem which provides high-resolution hardware timers. Programmable timers, also known as *interval*

timer, are essential in a real-time platform, since they allow to trigger release events, and in general enable the implementation of the event driven scheduling algorithms. However, not all the system architectures provide this feature.

In order to reduce worst-case overhead, the release queue implementation uses an interval timer per release time instead of programming a timer for every job. Jobs that share release time are organized in a binomial heap to be efficiently merged into the ready queue (in $O(Log(k))$ time, where k is the number of releasing jobs). A release timer is then associated to each release heap and programmed to expire at the correct time. A reference to the release heap is stored in a hash table of release times to minimize insertion cost of new jobs. From the user point of view, this solution reduces timer interrupts collapsing coincident release events into the same timer handler. Finally, release and ready queues are protected by spinlock against concurrent updates, which may also be used to serialize scheduling decisions.

Four main operations allow to operate on the `rt_domain`. They allow:

- `add_release()` to add a job to a release heap
- `add_ready()` to add a job to the ready queue
- `take_ready()` to remove the highest-priority job from the ready queue. The currently scheduled jobs must not be enqueued in the ready queue
- `peek_ready()` to return the highest-priority job in the ready queue without removing it. It is used to check whether a preemption is required

It is worth noting that to order tasks in the ready queue a comparison operator has to be defined, since the job priority depends on the scheduling algorithm (i.e.,EDF based schedulers use the deadline to order jobs).

The design of global and partitioned plugins mainly differs in the use of the `rt_domain`. Global plugin defines one shared `rt_domain`, declaring it in the global space. On the contrary, partitioned (or clustered) scheduler plugin defines a private `rt_domain` for each processor (or cluster). This separation is required so that jobs are merged into the appropriate processor-local (or cluster-local) ready queue. That is, `rt_domain` cannot be shared among multiple instances of priority-driven scheduling, rather, clusters are defined by the shared `rt_domain`. In this sense, the design space of the semi-partitioned approaches, like the RUN algorithm, is not well defined and leaves several degrees of freedom, as observed in the implementation Chapter 2.

Migration support

Task migration is required by both global and semi-partitioned algorithms, but poses considerable challenges due to the Linux kernel per-processor design.

From the scheduling point of view, a process migration always involves a target and a source processor. There are two basic migration models: the *push* and the *pull* migration model. Both these models are implementable in LITMUS^{RT}. In the push model, a source processor pushes waiting processes to the target processor, then sends it an IPI[1] to signal that a schedule operation is required. In the pull model, the target processor pulls backlogged processes from the source. As showed in [Bastoni *et al.*, 2011], the push model is more efficient than the pull one.

As explain above, Linux assumes that a currently executing process is assigned to the processor runqueue. In order to make a process migrate from a runqueue to another, both the runqueue locks have to be held. However, Linux requires the local lock to be dropped before the other lock is acquired. This creates great complications, since the state of all processes may change while the local runqueue lock is dropped. Moreover, processes cannot migrate instantly due to stack issues, as described in [Brandenburg, 2011].

LITMUS^{RT} solves these problems and embeds the solution in the scheduling class code to greatly simplify the implementation of global algorithms. From a user point of view, any process can be assigned to any processor at any time.

1.4 Contribution

In this thesis, we provide the first solid implementation of RUN with the twofold objective 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).

1.5 Structure

The Chapter 1 states the problem. It starts providing a background on the real-time scheduling, with some basic concepts and definitions. Then it presents the semi-partitioned approach to the real-time scheduling problem on multiprocessor. The challenge taken in this work is then discussed, first resuming some underlying properties of the RUN algorithm, then providing an overview of the LITMUS^{RT} testbed used to implement it. Chapter 2 explains our solution which is extensively evaluated in Chapter 3. Finally, in the Chapter 4 are drawn the conclusions and the open problems which remain to be investigated.

[1] The Inter-Processor Interrupt (IPI) is a mechanism by which a remote processor (target) can programmatically be notified by a source processor, in order to cause the invocation of the remote scheduler.

Chapter 2

Proposed solution

The implementation of semi-partitioned approaches are likely to require a much more complex structure, on the score of the fact that they do retain characteristics of both global and partitioned approaches. Actual implementations in [Bastoni *et al.*, 2011] show that there is no a unique conceptual model by which a semi-partitioned algorithm can be designed. In fact, it may strongly depend on the way tasks are allocated to processors. Therefore, only a systematic description of typical overhead related issues suffered by an algorithm is provided.

The description of RUN in [Regnier, 2012; Regnier *et al.*, 2011a], though detailed, leaves room for some implementation choices and raises few practical issues. A tentative implementation has been sketched in [Chishiro *et al.*, 2012]. The extremely brief description therein reported, however, seems to diverge from the strict formulation of RUN in that it simply remaps the reduction tree to static table-driven scheduling. In our implementation, instead, we stick to the original formulation of RUN as we deem the table-driven approach to be too rigid, incapable of accommodating any release jitter.

In the following, we present our implementation of the algorithm and point out the practical issues we encountered.

2.1 Design principles

The whole principle of RUN is based on two assumptions: **(i)** independent and fixed-rate tasks, and **(ii)** the accordance between on-line scheduling and off-line reduction phase. At the moment, assumption (i) does not require particular attention since our implementation is based on a periodic and independent task model that is represented by both the fixed-rate task model and the LITMUS$^{\mathrm{RT}}$ platform. The assumption (ii), instead, highlights the fact that the reduction tree is

essential to take online scheduling decisions. For this reason, much effort has been dedicated to representing the reduction tree and determining how to handle it for our scheduling purpose.

Before explaining our solution, we describe the actual scheduling model implied by the RUN principle. It is clear, recalling the Chapter 1, that the RUN schedule is achievable by composing two different levels of scheduling. One level is given by applying the scheduling rules upon the reduction tree to determine which tasks have to be allocated to the processors. These schedule decisions can be considered *global* since they can make tasks migrate from a processor to another. On the other hand, once a task is allocated, it has to be locally scheduled together with other tasks allocated on the same processor. In this case we make *local* scheduling decision since it can be used a uniprocessor scheduling algorithm such as EDF.

The level-0 servers represent the boundary between the global and the local level. Once an allocation is determined for each of the level-0 servers, we just have to apply the local scheduler among the aggregated tasks, in according to the server notion given in Chapter 1. In other words, starting from an updated state of the reduction tree, it is first determined an allocation of the level-0 server, which corresponds to the circled leaf nodes of the tree, then it is performed the local processor scheduling.

This scheduling approach is also important to characterize the origin of the scheduling events. The *release* and the *completion* of a job are the only events that require to take scheduling decisions. According to [Regnier *et al.*, 2011a], the release is a global event since it has to update the tree allowing to determine task allocation. In fact, every time the tree is updated, it may cause some tasks to be reallocated. On the contrary, the completion is clearly a local event since it just have effect to the local processor where the task is allocated. The event characterization helps to determine "who", "how" and "when" operates upon the scheduling entities.

Resuming, there is a scheduling flow that starts at the global scheduling level in consequence of one or more job releases. The tree is then updated in according to the RUN rules, which potentially determine a new server allocation. Each allocated server selects a task for executing until its completion or until the server is removed from the processor.

2.2 Implementing the reduction tree

Implementing the reduction tree is the main challenge we have focused on. The main problems we have addressed are: how to represent the nodes, how to update it, and how handle global events. Although no strict requirement is set by the algorithm on the way a reduction tree should be implemented, performance considerations suggest to adopt concise and fast data structures. In this first implementation of the

algorithm no particular attention has been paid on the way tasks and servers are packed together to form the reduction tree, and a simple *worst-fit* packing policy has been followed.

It is important to note that the reduction tree is a recursive data structure. Each node, when activated (circled), performs a scheduling activity with respect to its children nodes. At the same time, a node may be scheduled from its parent, if it exists, according to the server and client relationship described in Chapter 1. Therefore, in the general case, a node is a uniprocessor subsystem which state is updated at the *release* and the *completion* events of a children node. The release event adds one or more child nodes to the ready queue, on the contrary, the completion removes from the queue the node which job has completed. Both events change the ready queue and therefore a reschedule is required. Since the tree is recursively built, once an update occurs to a node S of the tree it has to be recursively propagated down to the sub-tree rooted in S.

LITMUS$^{\mathrm{RT}}$ offers the powerful abstraction for the real-time scheduling domain, termed `rt_domain`, which comes with a generic and reusable implementation of a scheduling context, comprising task queues and timers. The structure of the reduction tree, reported in Figure 1.5, and the implied hierarchy of run time entities seem to naturally suggest to associate an `rt_domain` to each server (either primal or dual): in 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 somehow independently, just following EDF rules. Unfortunately, the `rt_domain` implementation is unable to make recursively schedule activity to other `rt_domain` structures. It can only schedule actual tasks which are defined by the `task_struct` component. Only after a completely redefinition of the `rt_domain`, it may be exploited to build the reduction tree.

Moreover, we observed that the reduction tree is somehow redundant in that it provides both primal and dual representation of a server whereas just one would suffice. Hence, we use a compact representation of the reduction tree, which models just the dual nodes of the original tree and is based on a simple *left-child right-sibling* binary tree [Cormen *et al.*, 2001]. To improve on the bottom-up traversal of the tree each node is also augmented with a backward link to its father node.

Figure 2.1 hints at the implemented data structures. Each node in the tree just holds the information necessary to take the scheduling decisions: the earliest deadline among the children nodes, the current (updated) budget to simulate the node execution time and a flag indicating whether the node is currently selected for execution (circled).

In addition, an interval timer is assigned to each node in order to keep track of the execution budget consumption. Whenever a node selects a child node for

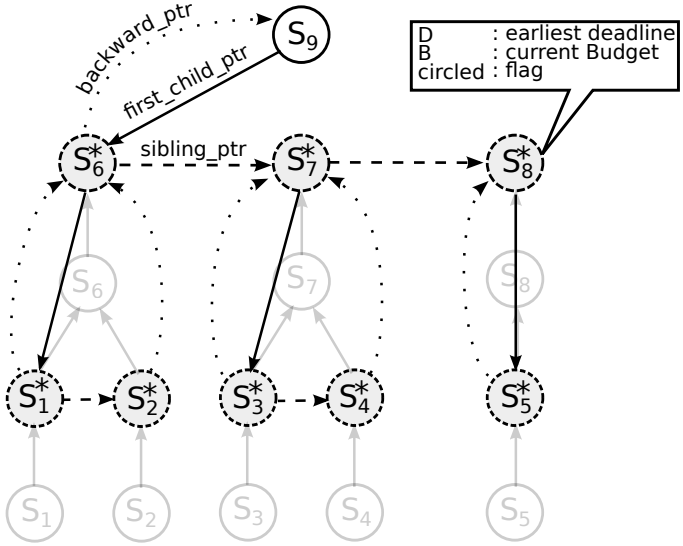


Figure 2.1: Reduction tree data structure.

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 a schedule point of view, a budget exhaustion corresponds to the job completion event in a subsystem. According to [Regnier *et al.*, 2011a], such an event is termed *virtual* since is produced at a level $k > 0$ of the reduction tree. In order to avoid confusion, we use the term budget exhaustion when we refer to the tree completion events; on the contrary, we say job completion when we refer to the local scheduler.

We are aware that our representation of the reduction tree as a binary structure does not behave optimally with respect to EDF policy. In fact, since sibling nodes are organized as a linked list, finding and selecting the child node with earliest deadline requires to traverse the whole list. To this purpose, an ordered heap would probably guarantee better performance.

However, we should also take into account that the scheduling tree belongs to the global level of scheduling and it is constantly shared among all processors. This means that the reduction tree may somehow migrate from one processor to another: on this account, the pithier the data structure, the better.

The reduction tree implementation does not aim to improve the efficiency but to reduce the code complexity. Some design improvements are discussed in Section 2.5.

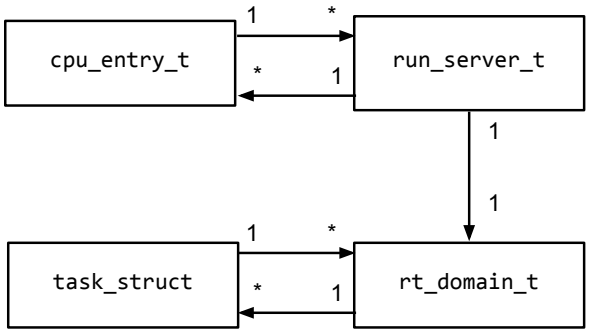


Figure 2.2: Scheduling components.

2.3 Implementing the scheduling domains and primitives

Although we pointed out the `rt_domain` structure is not suitable to implement the reduction tree, it seems reasonable to use it to implement the local scheduling level. As we said, the local scheduling is performed inside the level-0 server, once it is allocated to the processor. Thus, a possibility is to map an `rt_domain` to each of these servers. However, this solution has the serious drawback of being 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 one of such events – being a global event that involves updating the whole tree – would incur an increasing amount of overhead.

The opposite approach, consisting in having a single global `rt_domain` for the whole tree, still fails to distinguish between global and local events. A job completion is, in fact, a local event that has to be handled locally by its. In addition, having global system-wide scheduling structures exposes to the well-known overheads of global algorithms.

We opted for a halfway 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 just exploited the `rt_domain` in a non-standard fashion. We used a global `rt_domain` just to handle all release events: no information is actually stored in its task queues. A local `rt_domain`, instead, is associated to each level-0 server, just to handle the ready queue. The global `rt_domain` allows to optimize and collapse simultaneous release events (whose effects are global in RUN), whereas the local ones allow to contain the scope of the scheduling data structures.

The Figure 2.2 gives a complete picture of the scheduling components. In addition to the reduction tree, there are three other data structures:

- the `cpu_entry_t` structure represents the local processor state. It is declared

by using the Linux per-processor allocation macro `DEFINE_PER_CPU()` to accommodate a level-0 server and to perform the local scheduling. It has to be correctly retrieved every time the scheduler is invoked on a processor.

- the `run_server_t` structure is the actual level-0 server implementation. This structure can migrate among processors according to the reduction tree node selection and the allocation phase. On the one hand, it contains a `rt_domain` structure, on the other hand it is linked to the reduction tree in order to allow the tree update during task release events.
- the `rt_domain` is the well-known LITMUS$^{\mathrm{RT}}$ entity already described. Each globally released task is merged to the ready queue of the server it belongs to.
- the `task_struct` is finally the task. Once a task is scheduled, it is linked by the `run_server_t` for efficiency reasons. A task can only belong to one queue at the same time.

Finally, there is the global `rt_domain` structure. As we said, it allows to collapse all the simultaneous task release events into the timer handler included in it.

The online part of the algorithm relies on having the reduction tree always up-to-date with the deadline and budget information. Every time an event occurs in a processor, the kernel is invoked to handle it and is executed in the same processor. We distinguish three event types which are separately handled by the kernel:

- the *release* event
- the *completion* event
- the *budget exhaustion* event

The completion event is explicitly triggered by the completed job through the `sleep_next_period()` LITMUS$^{\mathrm{RT}}$ system call, unlike the release and budget exhaustion events that are timer triggered. Moreover, the two latter events are global and then operates in the global state, that is the reduction tree.

We have factorized two operations that are performed both at the release and at the budget exhaustion event: **(i)** the *tree update operation* (`r_tree_traversal` in the code), and **(ii)** the *server allocation* operation (`run_resched_servers` in the code). These operations exactly map the *scheduling function* and the *assignment function* defined in the Chapter 1.

In (i) we apply a slightly modified version of the iterative pre-order tree traversal [Cormen *et al.*, 2001] operating on the *left-child right-sibling* binary tree. This traversal strategy guarantees a correct node update, since it avoid to update a node whose parent is not yet updated. Therefore, we apply the RUN rules to the node

children assuming the node is already updated by its parent. Every time a node is selected (circled), the budget exhaustion timer of the parent node is armed. On the contrary, if a node is deselected (not circled), the parent timer is cancelled. Once the tree is completely updated, the leaf nodes indicates the level-0 servers that has to be allocated to the processor. Let t be the release instant, there are four possible scenarios:

1. A server circled before the time t is still circled after the time t.
2. A server circled before the time t is no more circled after the time t.
3. A server not circled before the time t becomes circled after the time t.
4. A server not circled before the time t is still not circled after the time t.

In (ii) we perform the actual level-0 server allocation to processors. According to these cases, we first determines the number of free processors (case 2) by passing all the m processors. Then, we pass all the level-0 servers to determine which ones belong to case 3, and we assign a free processor to each of these. Obviously, the number of free processors must be equal to the number of servers. Finally, we send an IPI from the processor which is performing the schedule to each processors whose server allocation is changed (case 2 and 3). We also send an IPI to the processor whose server allocation has not changed (case 1) but a task has been released, in order to invoke the local scheduler to perform a potential context switch.

To capture the release events, we implemented the `release_jobs_t` callback that is the handler of the release timer provided by the `rt_domain`. When a release occurs, the first operation is to merge all the released tasks to their corresponding ready queues contained in the server structures. Then, for each level-0 server whose ready queue has changed, we start the bottom-up update of the reduction tree. In this phase, we only update the earliest deadline and the budget values of the node as explained in the release event section. Then we apply the operation (i) and (ii) already described.

To capture the budget exhaustion event, we implemented the handler of the node timer. When the event occurs, we first set the budget of the node to 0 and then perform the operation (i) and (ii) as in the release case.

Finally, the local scheduler is implemented by the LITMUS$^{\mathrm{RT}}$ `schedule()` function, which has to select the next task to schedule in the processor. Unlike the other events, the `schedule()` function is aware of the processor on which it is running. In fact, we retrieve the processor state (`cpu_entry_t`) which contains the server allocated (`run_server_t`) and we obtain the next task to schedule by looking at the server ready queue. It is important to note that the `schedule()` function is invoked both every time a job completes and the processor receives an IPI.

2.3.1 The release event

Updating the tree is perhaps the most critical operation in RUN as, in order to keep the information in a consistent state, it must be executed upon each level-0 job release. Since the nodes in our tree just hold the information on the next deadline we actually only update the nodes along the path to the root if the release event causes a remodulation of the current sub-tree interval.

Specifically, the update process starts once a task release event occurs, considering the earliest deadline among the ready tasks in the same level-0 server. Suppose S_0 be a level-0 server and $H(S_0, t)$ its earliest deadline at the release time t, then many cases have to be considered to update a node $S_i \neq S_0$ along the path to the root:

1. $H(S_i, t) \leq t$. In this case, the earliest deadline of the node S_i has to be updated. Since the node S_i could have selected a child node S_j, we have to compare the S_j earliest deadline $H(S_j, t)$ with the S_0 earliest deadline $H(S_0, t)$.
   (a) if $H(S_0, t) < H(S_j, t)$ then the S_0 earliest deadline is also the earliest deadline for the server S_i, and then $H(S_i, t) \leftarrow H(S_0, t)$
   (b) else $H(S_j, t)$ is the earliest deadline for the server S_i, and then $H(S_i, t) \leftarrow H(S_j, t)$
2. $H(S_i, t) > t$. In this case, we have to update the node if and only if the S_0 earliest deadline is between the S_i release time $H(S_i, t-1)$ and the S_j earliest deadline.
   (a) if $H(S_i, t-1) \leq H(S_0, t) < H(S_i, t)$ then $H(S_i, t) \leftarrow H(S_0, t)$
   (b) else, we do nothing.

Case `2.b` takes its origin from the observation in[Regnier *et al.*, 2011b] that the deadline set of a server does not necessary include all deadlines of its clients, thus avoiding some unnecessary preemption points.

For each node whose deadline is updated, we also have to replenish the budget according to their utilization. In fact, the budget value corresponds exactly to the execution time of the node before its deadline.

The advantage of this solution is that we are able to keep the reduction tree updated avoiding duplications of the children earliest deadlines on each node.

2.3.2 The budget exhaustion event

The actual scheduling requires to update the reduction tree not only on release events but also upon budget exhaustion, as we said. The latter event, according to

our representation of the tree, corresponds to the expiration of a timer associated to a node *at any depth of the tree*. However, we do not need to reapply the scheduling rules on the whole tree but we may restrict our attention to the sub-tree rooted in the node where the expiration event has been intercepted. This shortcut is based on 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 [Regnier *et al.*, 2011a] (Lemma V.4). Suppose 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 deeply propagated to determine a context switch between exactly two leaf nodes in the sub-tree rooted in S.

2.4 Considerations

There are several considerations about our implementation of the RUN algorithm. Some are inherently attributable to the RUN model, others are attributable to LITMUS$^{\mathrm{RT}}$.

2.4.1 Task data structure

The task data structure (`task_struct` in the code) is the main scheduling entity. The offline reduction aggregates several tasks into the servers and this characterizes the RUN algorithm.

In order to determine to which level-0 server a task is aggregated during the online phase, we have augmented the default implementation of the task structure (`task_struct`), which keeps a reference to the correspondent level-0 server determined during the reduction phase. This reference is essential since the LITMUS$^{\mathrm{RT}}$ callbacks operate to the task struct, but all the scheduling events have also to update the server structure. For example, during the release phase, we have to merge the task to the correct server ready queue, or remove it at completion.

In order to link a task to the correspondent server, we enumerate it by an id, during the reduction phase, which is also used to admit or reject the task into the system.

2.4.2 Concurrent access to the global state

The RUN scheduler relies on keeping the tree up-to-date, but since it is global it has to be protected against concurrent updates. In order to guarantee the coherence of the scheduler state, we use a single and global lock. Every time a scheduling decision is needed, the lock has to be acquired serializing all the scheduling decisions. It is important to note that we acquire the lock also to preform local scheduling, since

a server may be removed from a processor while it is executing. We are aware that we reduce the concurrency of the kernel, but simultaneous scheduling events are very rare to justify the simplicity of our solution. In the future, we can investigate the possibility to use two types of lock: one for the global scheduling and one for the local scheduling.

2.4.3 Synchronous release instant

RUN relies on allocating the workload according to defined intervals. An interval is determined by two consecutive job deadlines. The problem arises when the first job release is considered, that is the notional instant `0`, which represents the start of the schedule. This instant is assumed by RUN.

The synchronous release is not activated by default in LITMUS$^{\mathrm{RT}}$, since tasks can be sporadically added to the system and they immediately become eligible. LITMUS$^{\mathrm{RT}}$ provides a synchronous release mechanism, whose details are described in Appendix A. Tasks are queued to a logical barrier once they are initialized and admitted into the system. Then, the barrier can be released by the `release_ts` tool. In order to determine the precise release instant, that is the notional instant `0`, we intercept the event related to the barrier by overriding the default behaviour. However, once the event is captured, the current system timestamp can not be considered because it may be inaccurate since it is taken slightly after the release time. To circumvent this problem, we use the time coming from the `release_ts` tool once it releases the barrier.

2.4.4 Overlapping events

We say that two or more events are overlapping if (i) they are global events and (ii) they occur simultaneously from two or more different timers.

Overlapping events are particularly annoying since they may cause unnecessary updates of the tree increasing the overhead. Moreover, the execution parallelism is reduced due to the need to acquire the global lock in order to handle them.

We have identify three situations during which overlapping events may occurs. The first, as already explained, is related to the simultaneous release of tasks which are aggregated into different servers. In this case, we collapsed the release event by merging each server release queue to a single global queue.

The second case happens every time a budget exhaustion event occurs on a node S, which at the same time should receive a replenishment caused by a release of a descendant task. In order words, the node S notionally executes at *zero-laxity*. In this case, the handler of the timer that triggers the budget exhaustion event may

execute just before or just after the handler of the release timer, requiring a fine-grained management. In order to avoid this behaviour, every time the timer of a node is armed, we check if the budget exhaustion time overlaps the release time. Hence, we can avoid to arm this timer since the tree will be, in any case, updated by the handler of release timer.

Finally, the third case happens when multiple budget exhaustion events are triggered simultaneously. Since this case can not be easily solved unless making several traversal upon the tree, we opted to leave unsolved.

We believe that overlapped events can be effectively removed by a different implementation design, as suggested in the enhancement section, since they are not intrinsically related to the RUN algorithm. On the contrary, the reason is related to the number of timers and therefore can be considered a design issue.

2.4.5 Job migrations

Once the tree is updated, RUN says nothing with respect to the actual assignment of tasks to processors. We should try to avoid unnecessary job migrations: 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. It is worth to note, however, that RUN allows only 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 [Bastoni *et al.*, 2011] this kind of migrations incur by far less overhead as compared to dynamically computed migrations, mainly due to global scheduling structures and the corresponding locks.

2.4.6 System workload

The full utilization requirement set by RUN is a prerequisite for duality to exist, and thus is crucial for guaranteeing the correctness of the algorithm. The requirement, however, is evidently unrealistic and requires some workaround: the authors themselves suggest the use of forced idle times or dummy tasks [Regnier *et al.*, 2011a] to cope with (i) variable job execution times and (ii) low utilization.

With respect to (i) we simply exploit the structure of $\mathrm{LITMUS}^{\mathrm{RT}}$ that allows native tasks to execute when no litmus task is ready. Within the scheduling domain of a level-0 EDF server, if a job J completes its execution before $J.r + J.c$ then either we execute the next EDF job in the server, if any, or just do nothing, thus allowing the execution of native tasks. Advancing in the local schedule, in fact, does not impair the overall schedule: on the contrary, tardiness is strictly banned by RUN as missing even a single deadline has severe repercussions on the whole system. When it comes

to (ii), instead of further increasing the task population, we would pragmatically prefer to exploit the available slack time as a buffer for run-time kernel overheads, by assigning proportional shares of the system slack to each tasks. It would be interesting to evaluate more sophisticated approaches to guide the slack allocation as it may affect the reduction process. Regardless of the overall utilization, in case of extremely low population, when there are less tasks than processors (i.e., $n<m$), then the reduction has to be conducted as if there were just n processors.

2.4.7 Off-line reduction phase

Although the construction of the reduction tree seems quite similar to the preliminary packing phase of partitioned and semi-partitioned algorithms, it's actual implementation might 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, actually correspond to processors.

As 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 (according to the tree) has been defined. A serialized version of the tree is given in input to the plugin so that it can be reconstructed within the kernel in a sort of a pre-initialization phase, and thus before admitting any task in the system. This procedure is described in the following section.

2.4.8 Kernel space reduction tree

The reduction tree start as an offline information, but is necessary to make online scheduling. Two main approaches are available to represent the tree inside the kernel. The first approach, is to off-line embed the tree to the source code of the RUN plugin. It requires the system rebuilds every time the tree has changed. Therefore, it is clamorous time-consuming operation by our purpose. The second approach, we have adopted, is to build the tree online during the system initialization phase. However, this is slightly more complicated since the user space and kernel space use different memory areas in Linux [Torvalds, 1994]. We introduced an additional system call that allow us to clone the tree from the user space to the kernel space, by copying one node at a time. After selecting the RUN scheduler, and before starting an experiment, the tree has to be build in the kernel. Therefore, a failure during this phase may definitively compromise the scheduler execution.

2.5 Enhancements and alternatives

2.5.1 Allocation approach

The solution presented so far, broadly follows the indications given in [Regnier *et al.*, 2011a]. Global events are generated on the tree by using an interval timer for each node that allows to trigger the budget exhaustion event and therefore to change the allocation to the level-0 servers. Once a level-0 server is allocated, it executes until a future global event removes it.

An in-depth comprehension of the RUN principle (e.g., the role of the reduction tree and the dual scheduling) led us to a slightly different formulation of the problem. Considerations in the Chapter 1 suggest that the dual plays a fundamental role on the job allocation. Each time a job can not be completely allocated to a processor, it is fragmented into two or more parts whose find a processor allocation thanks to the dual that "reserves" exactly the required job execution time. Hence, the job allocation can be determined according to the reduction tree.

At any level of the tree, a circled node, primal or dual, means that the node is executing. The set of the circled nodes, in a given time, is the actual information we needed to determine the job allocation. Since there are exactly m level-0 servers executing at the same time, there can be at most m context switches due to the global scheduling. This suggests that the global event can be attributed to the processor, when the level-0 server is allocated, rather than be attributed to a node on the tree. Moreover, a context switch in the tree which involves two sibling nodes is propagated downwards to tree involving exactly two level-0 servers, as showed in [Regnier *et al.*, 2011a]. Once the server is allocated to the processor, the correspondent timer is armed according to the exact server execution time assigned to that processor. For this reason, we believe that timers can be moved from the nodes of the tree to the processors with several benefits.

Unfortunately, determining the actual execution time once a server is allocated is not trivial, but we believe the reduction tree is sufficient to determines this information.

2.5.2 Reducing the release queue contention

Having a single global `rt_domain` is a point that deserves to be discussed. We have already introduced the problem that arises by using one `rt_domain` from each level-0 server: every time a task or a group of tasks are released in a level-0 server, the tree has to be updated, leading to have several instants during which the tree is repetitively updated. This causes an unacceptable overhead on release.

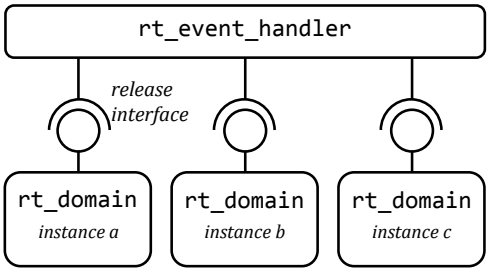


Figure 2.3: `rt_domain` interface.

In order to improve this situation, we solved it by collapsing all the release events exploiting a single global `rt_domain`. This solution, however, may also incur an high overhead since all the system tasks are queued in a single global release queue. Since the release may happen in any processors, the global `rt_domain`, on which the release queue is hold, may suffer the continuous migration from a processor to another. In this case, the longer is the queue the greater the overhead.

Reasoning about the reduction, we can observe that is not always necessary to have a single global queue. In the Chapter 1, we gave a formal definition to the reduction which states that the task set is partitioned in subsets of tasks, denoted Γ_i. Tasks in a partition are isolated respect to the other partitions and one unit-server exactly corresponds to one partition and, thus, to one reduction tree. Therefore, it is possible to use a release queue, then an `rt_domain`, for each of these partitions.

There are two important consequences to consider. For each subsystem:

- the release queue is shared only by tasks in the partition
- in the average cases the release queue is shorter
- in the average cases the reduction tree is smaller

All these consequences may reduce the overhead, still collapsing release events of tasks in the partition into a single handler. Moreover, a great advantage in adopting this approach is that we can use a lock for each of these "global" `rt_domain` to protect both the release queue and the reduction tree. However, this requires to redefine several data structures in order to accommodate multiple instances of the reduction tree and global `rt_domains` besides reimplementing the tree copying procedure described above. In our solution, we just support a single global reduction

tree, which formally is equivalent as noted in [Regnier *et al.*, 2011a], but greatly simplifies the implementation stage.

We also investigate a different possibility, which basically aims to redefines the `rt_domain` interface. In this approach, the interface should allow to "export" the release timer from inside the `rt_domain` to the outside and to share it among several `rt_domains`, as depicted in Figure 2.3.

An event collector (`rt_event_handler`) allows to decouple the timing needs coming from several `rt_domains`. Each release timer can be moved to the event collector, but leaving at the same time the release queue into the `rt_domain`. Then, globally defining the event collector, it is possible to collapse simultaneous events into the same handler. At this point, the `rt_domain` can be used to exactly represent the level-0 server.

Chapter 3

Empirical evaluation

Experiments consist of three phases, as depicted in Figure 3.1:

1. the creation phase during which a set of experiment is created. Task set are randomly generated varying their characteristics (i.e., the task utilization distribution, the total system utilization, etc.)
2. the execution phase during which execution traces are collected by feeding the same task sets to the three scheduling algorithms (RUN, G-EDF and P-EDF). It stands to reason that the performance of each algorithm depends on the task set features and the overall system workload in particular.
3. the analysis phase during which are generated and compared the plugin execution statistics

These phases are then repeated in order to refine results or to correct implementation related issues.

3.1 Environment setup

To guarantee a fair experimental process, 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 utilizations as we expect most important variations to occur then. Task utilizations, instead, 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%. Moreover, tasks were constrained to exhibit harmonic periods drawn from a uniform distribution in [25ms; 200ms] so as to keep the experiment manageable in bot size and complexity. We used the lightweight tracing facilities offered by LITMUS$^{\mathrm{RT}}$ to collect execution

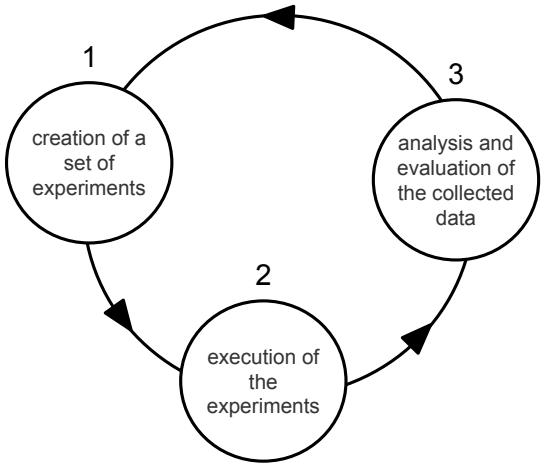


Figure 3.1: Experiment phases.

traces for the three algorithms, where each task set was executed for 30 seconds thus multiple times their hyperperiod.

Since we expect kernel overheads to also vary on the number of cores available, we performed three sets of experiments targeting respectively 4, 8 and 16 cores. Experiments were run on a dual-processor Intel Xeon E5-2670[1] system where each processor includes 8 cores running at 2.6 GHz and 20 MB of L3 cache. Hyper-threading and power management functions were intentionally disabled. A block diagram which is focused on the cache hierarchy and the core interconnection is reported in Figure 3.2.

This SMP platform exploits a very sophisticated architecture. Each processor (CPU0/CPU1) is composed by eight cores interconnected together according to a ring topology which allows to share the last level cache (L3), the memory controller and the QuickPath Interconnect bus. This bus enables the intra-processor communication and to enhance its bandwidth it is composed by four independent rings: a data ring, request ring, acknowledge ring and snoop ring. Finally, the L3 cache is divided into slices, one associated with each core although each core can address the entire cache.

We are aware that the cost of preemptions and migrations may be in large part determined by cache-related interference [Altmeyer and Burguiere, 2009; Bastoni *et al.*, 2011]. However, in our setting cache effects are actually negligible because tasks exhibit extremely reduced working sets (i.e., tasks simply execute an empty loop). Therefore the reported overheads are not representative of cache-related effects: we defer the quantification of such overheads to future work.

[1] Specifications at `http://ark.intel.com/products/64595/`

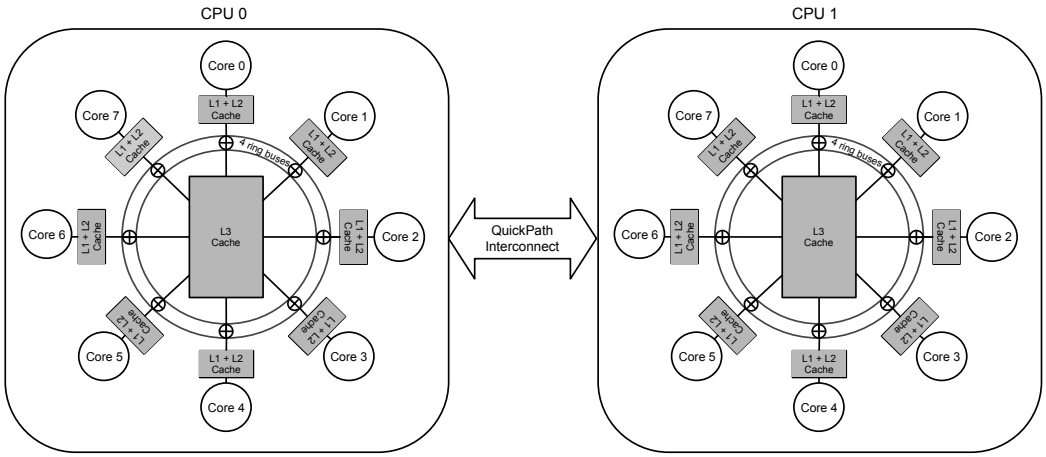


Figure 3.2: Hardware architecture.

3.2 Objectives

An extensive evaluation by simulation of RUN has been reported in [Regnier, 2012; Regnier *et al.*, 2011a]. The cited works show that RUN significantly outperforms other optimal multiprocessor scheduling algorithms in terms of incurred preemptions and migrations. As observed in [Silberschatz *et al.*, 2008], 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 (i.e., viability) and exhibit actual kernel overheads that may not be always evident. The proposed implementation of RUN on top of LITMUS$^{\mathrm{RT}}$ demonstrates that the algorithm can be actually implemented just relying on standard operating system support, and enables us to conduct an extensive empirical evaluation of the algorithm.

In our experiments we were interested in evaluating the scheduling algorithm with respect to the interference it causes to the system rather that finding its empirical utilization caps. For this reason, we set our main focus on assessing the kernel overhead in terms of both the cost of its primitives and the number of preemptions/migrations incurred. Accordingly, we did not want to evaluate RUN against other optimal algorithms: we wanted, instead, to compare it against P-EDF[2] and G-EDF (both included in the LITMUS$^{\mathrm{RT}}$ 2012.3 release) as representatives of global and partitioned algorithms. The choice of P-EDF and G-EDF is not arbitrary as (i) RUN re-

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

duces 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 the G-EDF.

3.3 Results

It should be noted that single task utilizations includes a 5% slack, as a buffer for kernel and tracing overheads. Consequently also the reported overall system utilization is just nominal.

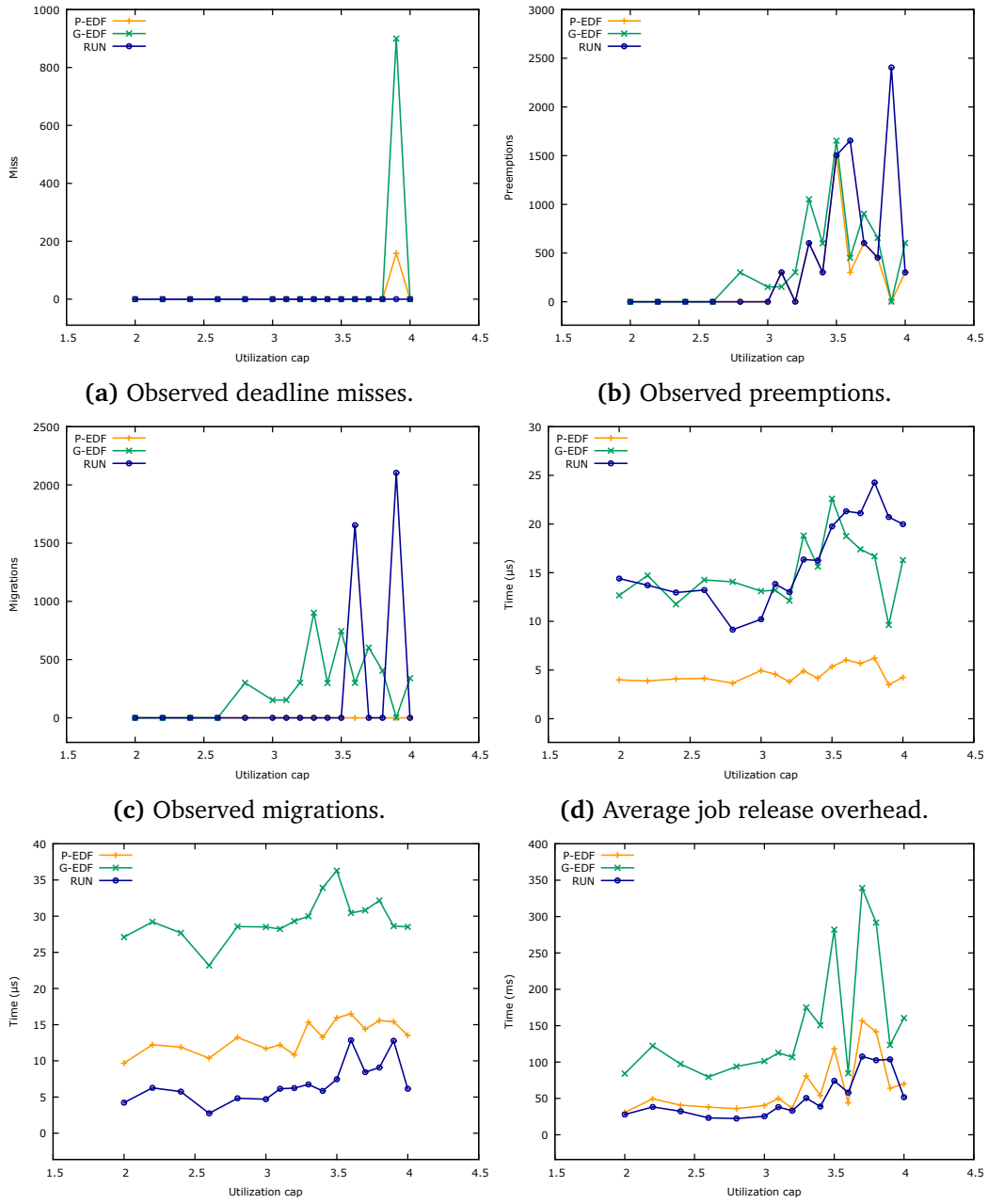
Although for completeness we have reported experiments performed on 4, 8 and 16 cores, these broadly confirm the same trend, as it can be observed in Figures 3.3 - 3.4, 3.5 - 3.6 and 3.7 - 3.8 respectively.

Experiments performed on 4 cores consist of a relatively small number of tasks and processors, therefore have produced quite coarsed grained results. We simply report them without providing an in-depth description.

Figure 3.5 reports the results observed on 8 cores (single processor). As shown in diagram 3.5a, RUN is capable of sustaining extremely high workloads, up to a theoretical 100% system utilization. P-EDF is penalized by higher utilizations as it cannot guarantee good solutions to the packing problem (problem cured by dual representations in RUN). Conversely, G-EDF already produces invalid schedules at medium utilization levels. We conjecture that the kernel overhead incurred by RUN is sufficiently low to fit into the provided 5% slack.

When it comes to kernel interference, the number of preemptions incurred by RUN (Figure 3.5b is quite similar to those observed for P-EDF. 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 fulfil the 100% utilization requirement. Excluding its invalid schedules, G-EDF incurs only slightly less preemptions than RUN; however, almost all such preemptions result in a job migration, as shown by comparing Figure 3.5b and 3.5c. On the contrary, RUN only incurs some migrations at extremely high utilizations and behaves worse than G-EDF only in those cases where G-EDF produces an invalid schedule.

The plain overhead incurred by each scheduling algorithm, instead, is reported in Figure 3.5d and 3.5e, showing the average execution time observed for the release and schedule primitives. It is worth noting that the cost of updating the reduction tree (quite limited in practice) is included in both primitives. As expected the use of local (per partition) data structures makes P-EDF outperform the other algorithms relatively to the cost of a job release. The fact that the cost of a job release in RUN is quite modest in comparison to G-EDF depends on the optimization we applied when



(a) Observed deadline misses.

(b) Observed preemptions.

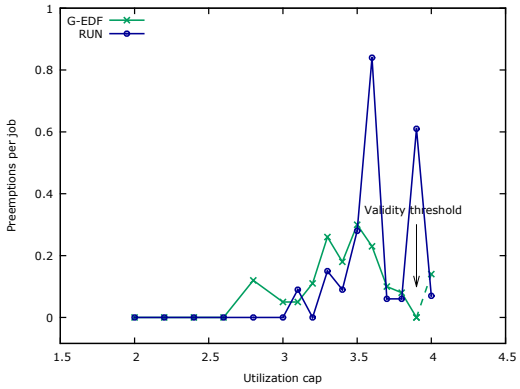
(c) Observed migrations.

(d) Average job release overhead.

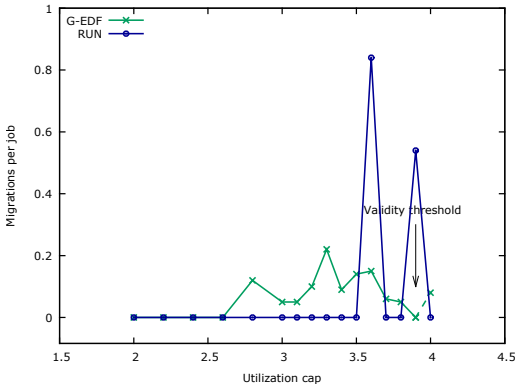
(e) Average dispatching overhead.

(f) Cumulative kernel overhead.

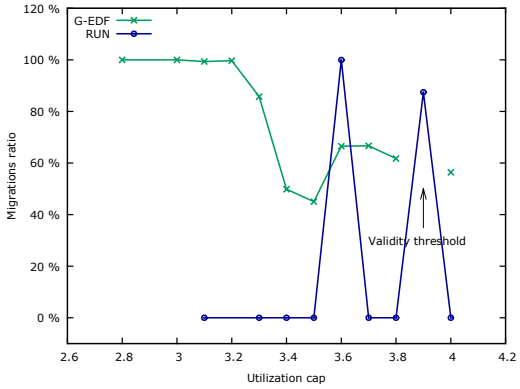
Figure 3.3: Experimental results on 4 cores under increasing system utilization.



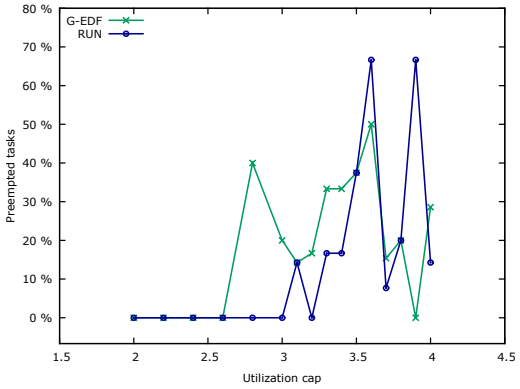
(a) Average preemptions per job.



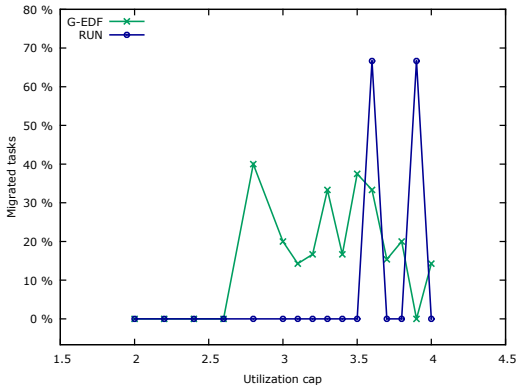
(b) Average migrations per job.



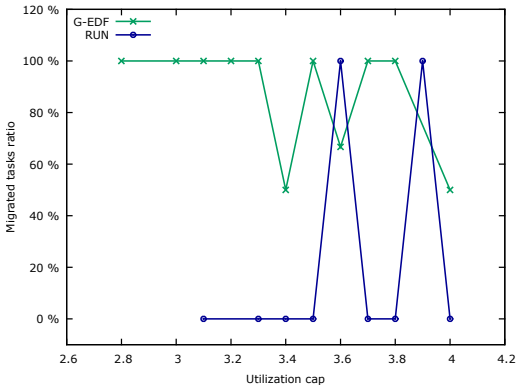
(c) Average migration ratio.



(d) Preempted tasks ratio.

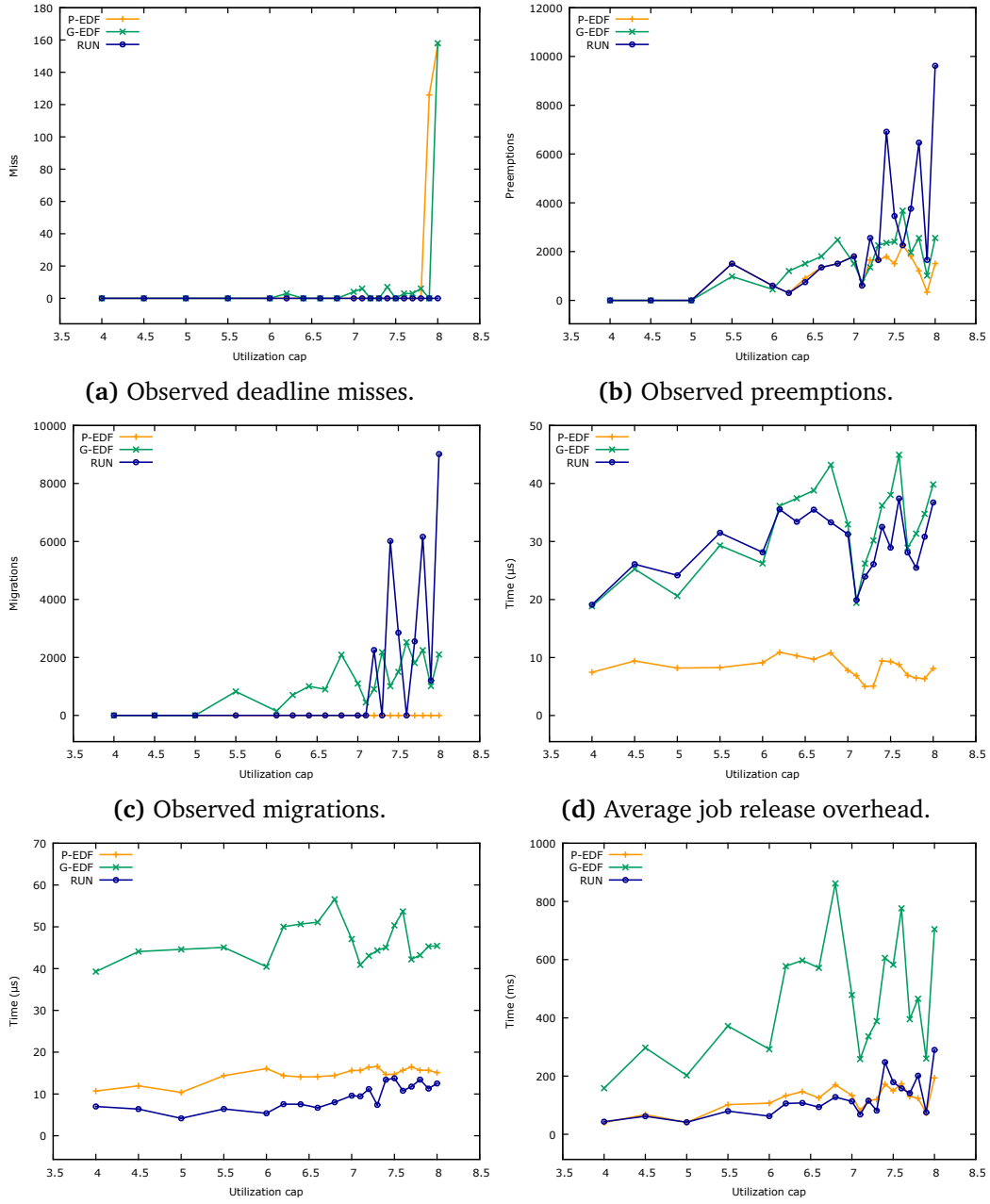


(e) Migrated tasks ratio.



(f) Migrated upon preempted tasks ratio.

Figure 3.4: Experimental results on 4 cores under increasing system utilization.



(a) Observed deadline misses.

(b) Observed preemptions.

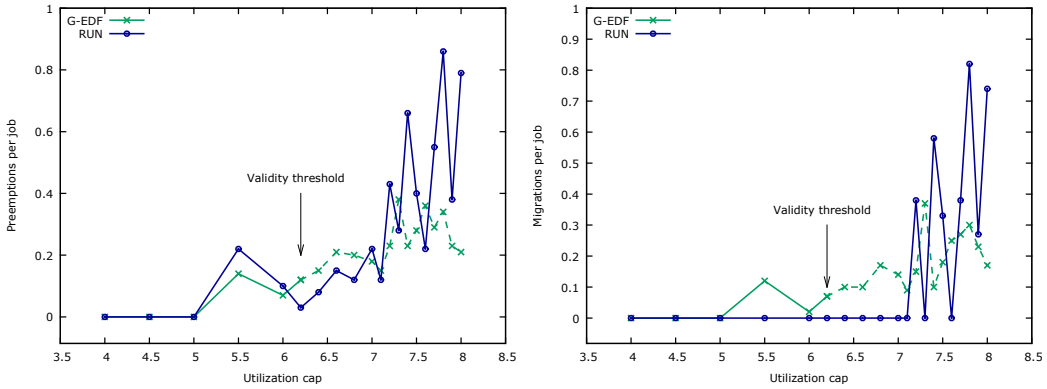
(c) Observed migrations.

(d) Average job release overhead.

(e) Average dispatching overhead.

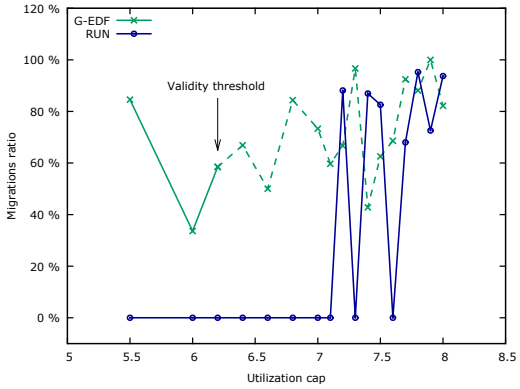
(f) Cumulative kernel overhead.

Figure 3.5: Experimental results on 8 cores under increasing system utilization.

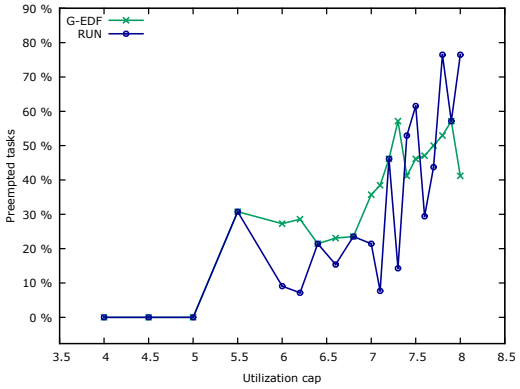


(a) Average preemptions per job.

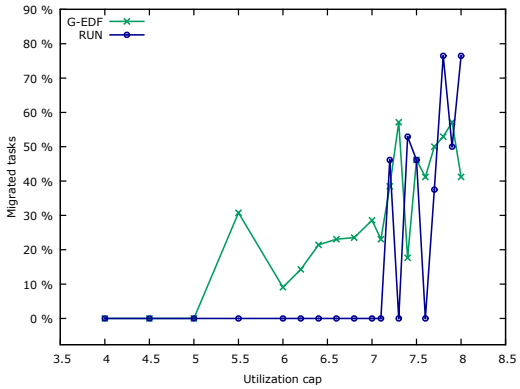
(b) Average migrations per job.



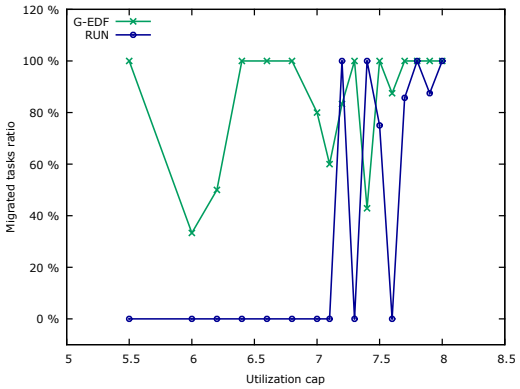
(c) Average migration ratio.



(d) Preempted tasks ratio.

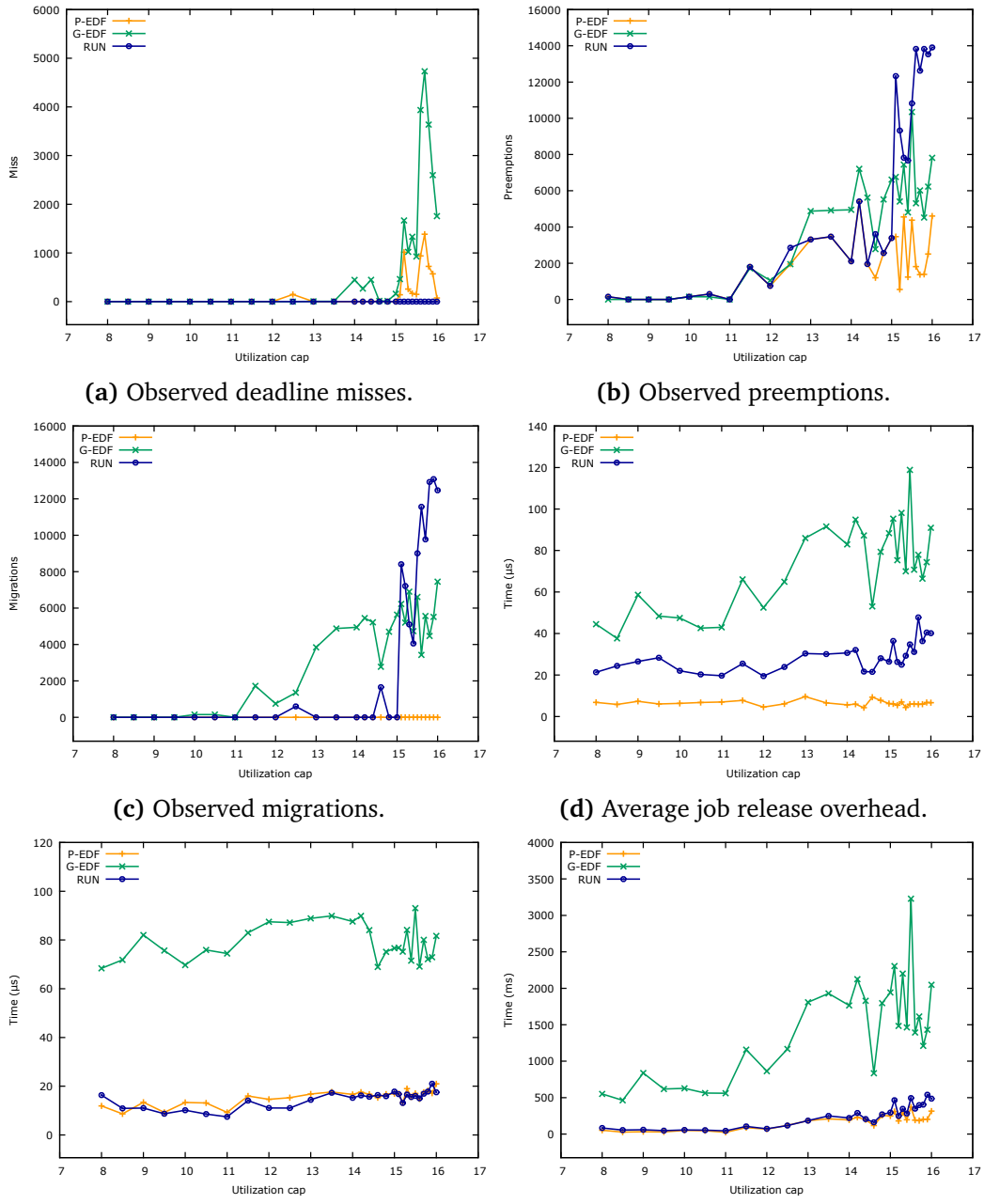


(e) Migrated tasks ratio.



(f) Migrated upon preempted tasks ratio.

Figure 3.6: Experimental results on 8 cores under increasing system utilization.



(a) Observed deadline misses.

(b) Observed preemptions.

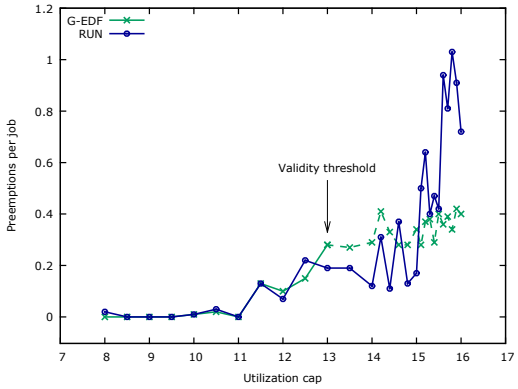
(c) Observed migrations.

(d) Average job release overhead.

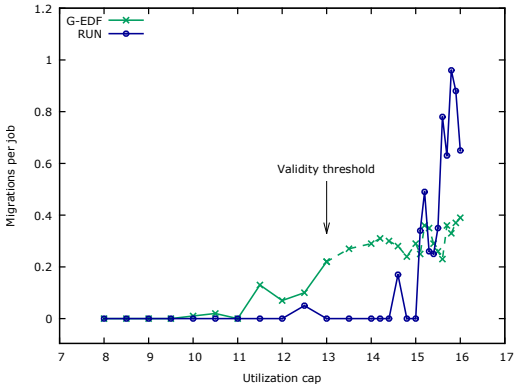
(e) Average dispatching overhead.

(f) Cumulative kernel overhead.

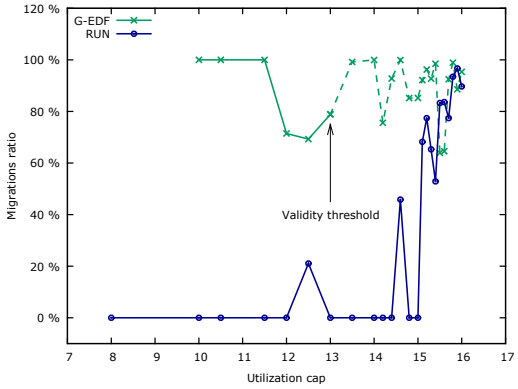
Figure 3.7: Experimental results on 16 cores under increasing system utilization.



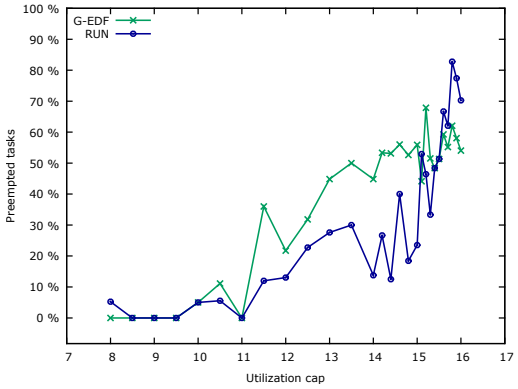
(a) Average preemptions per job.



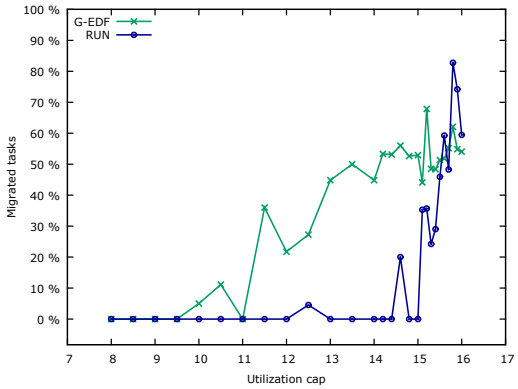
(b) Average migrations per job.



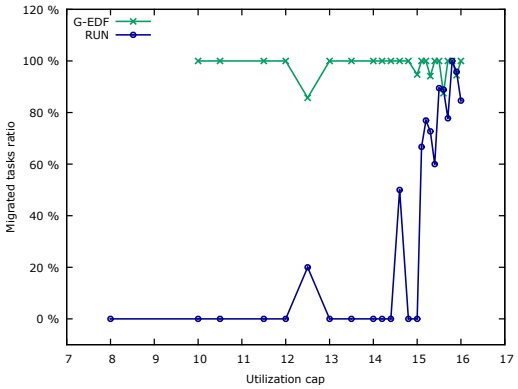
(c) Average migration ratio.



(d) Preempted tasks ratio.

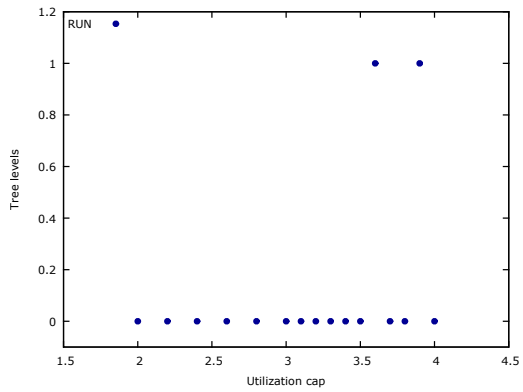


(e) Migrated tasks ratio.

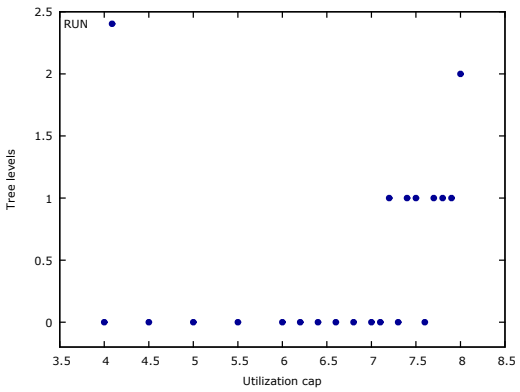


(f) Migrated upon preempted tasks ratio.

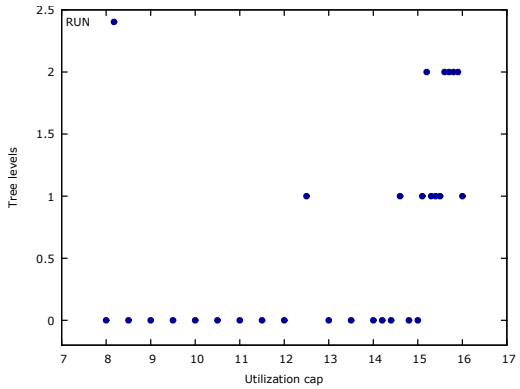
Figure 3.8: Experimental results on 16 cores under increasing system utilization.



(a) Observed reduction tree levels.



(b) Observed reduction tree levels.



(c) Observed reduction tree levels.

Figure 3.9: Reduction tree levels on 4, 8 and 16 cores (Figures 3.9a, 3.9b and 3.9c respectively).

handling simultaneously multiple job releases. 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 but uses a separate queue for each level-0 server (as explained in Chapter 2). The cost of a schedule in RUN is thus similar to that of P-EDF. The effects of this favourable condition is an extremely low cumulative overhead, as shown in Figure 3.5f.

We also observed the amount of interference suffered by each job in the average case and by tasks (Figure 3.6). Figures 3.6a and 3.6b show the average number of preemptions and migrations observed per job under G-EDF and RUN. These diagrams mirror the cumulative behaviour reported in the previous set of figures. A more interesting insight can be drawn instead from Figure 3.6c, reporting the ratio between migrations and preemptions, where a gap means that no preemption actually occurred. For low and medium utilization levels RUN causes less migrations than G-EDF, which instead shows a high probability of migration. For higher utilization levels also the migration ratio in RUN increases: this could be seen as the price that has to be paid to guarantee task set feasibility. Figures 3.6d shows the percentage of tasks preempted at least once and Figure 3.6e shows the percentage of the task migrated at least once. These results aims to show the the behaviour of the algorithm respectively to the propensity of preempt and migrate tasks, in other words, how preemptions and migrations are distributed respectively to the task set. In Figure 3.6d in correspondence to a similar number of preemptions (per job or cumulative) we observe that in G-EDF this is produced by an higher number of tasks, compared to RUN. The reasons is that in RUN preemptions are mainly due to the aggregation of tasks in a server. For instance, suppose a high frequency task, hence with a small period, is aggregated to a task with a very long period. Applying EDF, the latter suffers the interference due to the more frequent task, causing potentially a great number of preemptions. Even more evident is the case of task that migrates, in Figure 3.6e. Although G-EDF caused more migration that RUN, these are distributed over a significantly number of tasks. The last Figure 3.6d reports the the ratio between the migratory tasks over the preempted tasks. We observed that in G-EDF almost preempted tasks are also migratory, on the contrary in RUN this is strictly dependent to the migration ratio.

Similar experiments were conducted on 16 cores, whose results are reported in Figure 3.7 and 3.8. In this case, we should consider that experiments were actually executed on two 8-core processors and thus kernel overheads also include inter-processor effects. This fact, however, does not invalidate the results. As in the 8-core experiments, P-EDF and G-EDF cannot always guarantee all jobs to meet their deadline (Figure 3.7a): again, our comparison focuses exclusively on valid schedules.

The observations on preemptions and migrations, reported in Figure 3.7b and 3.7c respectively, confirms the trend from the 8-core experiments. From Figure 3.7b we observe that RUN generally incurs the same number of preemptions as P-EDF and always less than G-EDF. As the only relevant exception, at utilization 12.5 RUN seems to suffer more preemptions than both the other algorithms. However, we also observe that in that case P-EDF incurs some deadline misses (see Figure 3.7a) which is explained by the fact that P-EDF was unable to find a perfect packing. In this case RUN also did not find a perfect packing and was forced to resort to a 2-level reduction tree which is a potential source of preemptions and migrations. As for migrations, instead, RUN always outperforms G-EDF (Figure 3.7c).

Overhead incurred by the scheduling primitives is reported in Figures 3.7d- 3.7f. The three diagrams confirm the trend observed in the 8-core experiments: the overall cumulative overhead suffered by RUN is quite low and almost assimilable to P-EDF. Moreover, comparing this series of diagrams with their 8-core counterparts, we also notice that both P-EDF and RUN scale exceptionally well with number of cores whereas the overhead suffered by G-EDF explodes.

Figures 3.8a and 3.8b show the average number of preemptions and migrations observed per job under G-EDF and RUN. We obtained similar results observed in the 8 cores experiments, and therefore, we can confirm the above considerations. Also, Figure 3.8d and 3.6e show that in G-EDF many tasks are affected by preemptions and migrations compared to RUN, confirming the effect of the task aggregation of RUN.

Finally, in Figures 3.9a, 3.9b and 3.9c are reported the reduction levels of the tree generated by this set of experiments. We observed that more processors are considered more is the probability to produce 1 or 2 reduction levels.

Chapter 4

Conclusions and future work

We presented the first solid implementation of the RUN multiprocessor scheduling algorithm on top of the LITMUS$^{\mathrm{RT}}$ test bed. 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 costs and practical viability. Our experimental results suggest that depending on the underlying implementation RUN may exhibit quite moderate kernel overhead, almost similar to that of a partitioned algorithm, such as P-EDF. The tree update does not introduce a significant overhead since it is very compact; the release overhead is, in fact, mainly due to the the use of a global release queue. In addition, our experiments also confirmed that RUN minimizes the number of migrations with respect 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 confirmed that the number of preemptions and migrations in RUN heavily depends on the way the off-line reduction tree is built, including but not limited to, the packing heuristic adopted. We plan to further investigate practical means to build *good* reduction trees that can further reduce the number of preemptions and migrations. As an orthogonal line of research, instead, we are studying whether and how RUN could accommodate (i) *real* sporadic tasks, whose job release time is not known beforehand; (ii) collaborative tasks, whose execution is constrained by precedence relations or a shared resources. We briefly present these problem in the following.

We consider the LITMUS$^{\mathrm{RT}}$ platform suitable for the objectives of this work. It allowed us to implement RUN with a little effort of time and to perform several experiments, also using different computing platforms. The main difficulties were due to the relatively poor documentation and the lack of examples about the plugin

development. In particular, very little information, often disorganized, describe the semantic of the scheduling primitives and data structure invariants. Tools through which perform the analysis of data collected during the experiments are also missing. This has required to implement them from scratch.

4.1 Open problems

The whole principle of RUN is based on two assumptions:

- independent and fixed-rate tasks
- the accordance between on-line scheduling and off-line reduction phase of the task set.

These assumptions are clearly ideals since are made in order to simplify the RUN model and reduce its viability in real-world solutions. Task independence assumption, in particular, should be taken into account since in several situations is required the collaboration among tasks. Moreover, strictly periodic tasks cannot model all the system design requirements: a task that serves an external sporadic interrupt, for example, should be modelled as a sporadic task. These problems are briefly described below.

4.1.1 Collborative tasks

A collaborative task is commonly a task which execution is constrained by:

- a precedence relation with the execution of other tasks
- one or more logical resources shared with other tasks

We believe that a viable approach is to selectively and transparently remove the independence constraint without modifying the task model of RUN, eventually by acting on the offline phase.

Hence, defining at the beginning a task set containing collaborative tasks, processing it in order to satisfy the independence assumption and finally performing the off-line reduction phase, allows us to preserve the principle of RUN.

Formally, let C and I be disjoint sets of collaborative tasks τ^C and independent tasks τ^I respectively. Let $S = C \cup I$ be their union set. Let Γ be a *partition*[1] of the set

[1] Γ is a partition of a set X iff

$$(\varnothing \notin \Gamma) \ \wedge \ (\bigcup \Gamma = X) \ \wedge \ (\forall\, A, B \in \Gamma \wedge A \neq B) \Rightarrow A \cap B = \varnothing$$

C such that $|\Gamma| = k$ and let $\Gamma_i = \{\tau^C\}_i \in \Gamma, i \in [1,k]$, then $|C| = \sum_{i=1}^{k} |\Gamma_i|$. We say that Γ is *valid* if for all $\Gamma_i, \Gamma_j \in \Gamma \mid i,j \in [1,k] \wedge i \neq j$ are *independent*, that is no tasks in Γ_i collaborate with tasks in Γ_j. Since Γ_i can be represented[2] as a single fixed-rate task τ^R, given a task set S and determined a *valid* Γ, it is possible to create a task set T^R such that $\tau^R \equiv \Gamma_i$ or $\tau^R \equiv \tau^I$ and all the constraints of RUN are satisfied. It is important to ensure that the representation of Γ_i as a fixed-rate task does not introduce an additional load, in other words it does not have to increase the total load of the system or it may prevent the optimality.

The architectural model of RUN can be extended by a further level which handle collaborative tasks. Thus, the levels are:

1. A level of *global scheduling* where each task τ^R is allocated to a processor consistently with RUN

2. A level of *local scheduling* where each task τ^R is executed according to its priority compared to the same processor allocation tasks

3. A level of aggregated task scheduling where each task τ^C is executed according to its priority compared to the same aggregated tasks

A task $\tau^R \equiv \Gamma_i$ has a more complex structure than a task $\tau^R \equiv \tau^I$, consequently the context-switch has a much higher cost. By acting off-line and on-line, we have to ensure that scheduling events do not interfere with the execution of these aggregated tasks. The off-line phase allows us to selectively remove global events that may cause migration whereas the on-line phase allows us to remove local events that may produce preemption.

Resuming, three main problems are identified:

1. Isolate, before the off-line phase, the independent parts of the system from those collaborative to obtain aggregates of tasks that can be represented as a fixed-rate without causing an excessive waste of resources and to be the starting point for determining the reduction

2. Selectively remove migration of each aggregate of collaborative tasks after determining its reason by adjusting the off-line reduction

3. Manage the on-line phase of each aggregate of collaborative tasks by a proper scheduling algorithm placed between the global scheduler and the local one

[2] Γ_i can be represented by an interface of composition, so it forms a part of the system that consists of a dedicated scheduler as well as the collaborative tasks.

4.1.2 The sporadic task model

A fixed-rate task relies on the knowledge of the next event of release (deadline) to correctly allocate the workload according to the rate. The problem arises for the fact a sporadic task is characterized by a release event not known a priori but it always occurs after a minimum guaranteed distance from the previous ones. The worst-case arrival behaviour occurs when the release of a new job occurs exactly at the minimum guaranteed distance from the previous event of release. In this case, the task can be considered periodic and a schedule can be computed by RUN. Intuitively, a sporadic task whose release event occurs less frequently than its worst-case arrival time, for which its rate is guaranteed, may waste part of its allocated slot causing the failure of the schedule. Thus, the original formulation of RUN is then only able to schedule periodic tasks since the full-utilization assumption can easily fulfilled.

Appendix A

The LITMUS^{RT} environment

A.1 The LITMUS^{RT} plugin interface

The LITMUS^{RT} plugin interface consists of 13 functions that can be grouped into four classes of functionality, as listed in Table A.1. All these functions allow to specify operations in response to an event (i.e., release and completion of the job, scheduler invocation, etc.). In this section the term task is used instead of the term process for coherence to the real-time terminology.

Table A.1: List of methods in the LITMUS^{RT} plugin interface.

Method	Purpose
`schedule()`	Select next task to schedule
`tick()`	Trigger the scheduler at a quantum boundary
`finish_switch()`	Operate after the context switch
`release_at()`	Prepare future timed job release
`task_block()`	Remove a task from ready queue
`task_wakeup()`	Add task to ready queue
`complete_job()`	Prepare a job for the next period
`admit_task()`	Check if task is correctly configured
`task_new()`	Allocate and initialize task scheduler state
`task_exit()`	Free task scheduler state
`activate_plugin()`	Allocate and initialize plugin state
`deactivate_plugin()`	Free plugin state
`allocate_lock()`	Allocate a new real-time lock

All these functions are optional, with the exception of `schedule()` and `complete_job()`, this means they are not required to be implemented and are replaced by a default implementation.

There are two main scheduling functions to implement scheduling logic: `schedule()` and `tick()`. These are dedicated to event-driven scheduling and quantum-driven scheduling respectively.

- `schedule()` is the catch all scheduling function called whenever a task has to be selected for execution. Conceptually, it implements the context switch event, and the execution occurs in the same processor where the involved task is executing.

- `tick()` is called at each quantum boundary to handle periodic timer interrupt, becoming the central scheduling method in quantum-driven algorithm, such as PFair. If a preemption is required, then `tick()` can cause `schedule()` to be called by setting a rescheduling-flag to the currently scheduled task.

- After a context switch has occurred, the plugin is notified through a call to its `finish_switch()` function. Generally, this function is not implemented.

The following methods are related to the job lifecycle and state changes. The purpose of these methods is to track whether a task is eligible to execute. These methods add and remove tasks from the ready queue if needed.

- The method `release_at()` is called to notify future releases of the job. This is not suitable for periodic job releases, rather it is useful when a task requires a defined timed job release (e.g.,together with the barrier mechanism which allows to synchronize the first job release of the tasks).

- `task_block()` reports when a task is suspended. This function should remove the suspended task from the ready queue.

- When a task resumes, it is reported by a call to `task_wakeup()`. In this case, it must add the task to the ready queue, otherwise the task handler is lost.

- When a job finishes to execute, the `complete_job()` method is called through which the job is prepared for the next release and inserted into the release queue.

Finally, there are some functions which allow to define task and plugin life-cycle. Three functions perform admittance, initialization and cleanup activities of real-time tasks: `admit_task()`, `task_new()`, `task_exit()` respectively. Two methods are related to plugin initialization and cleanup: `activate_plugin()` and `deactivate_plugin()`.

It is very important to get the correct task status before deciding where it has to be enqueued since it may corrupt the system status.

A.2 The user-space interface, library and tools

The point of contact between the LITMUS^{RT} kernel and user space is constituted of a number of system calls and several virtual character devices. Moreover, there is a user-space library, named *liblitmus*, which implements system call stubs and several utilities and macros to simplify programming of real-time tasks. In this section all these parts are briefly described.

System calls can be grouped according to their functionality as follows:

- task definition: it allows to declare a process as a LITMUS^{RT} real-time task. Each parameter (e.g.,WCET, period, deadline, processor mapping) is copied in the real-time task structure (`rt_task`)
- job control: it allows to get the current job sequence number, wait for a job releases and signal a job completion
- system call overhead measurement
- creation, lock, and unlock of real-time semaphores operations: not interesting for our purpose
- synchronous task set release: it allows the first job of each task to be released at exactly the same time (the notional instant 0). To achieve high precision on releasing, LITMUS^{RT} defines a new system call (`wait_for_ts_release()`) because either *pthreads* barrier or the `nanosleep()` system can incur in a slightly time shifting issue. Moreover, it allows to detect whether each process is ready for real-time execution before any job starts. Each process that invokes this system call after finishing its initialization phase and transitioning into real-time mode is suspended until the synchronous release occurs. A non-real-time process monitor opens the barrier as soon as all real-time processes are ready to start, thus causing the synchronous task set release. By using the `release_at()` function, the pluging can capture this event, for example to set the absolute global time of release.

Besides exporting simple status information (e.g.,, the number of real-time tasks, the number of tasks ready for synchronous release, etc.) by extending Linux's standard `proc` file system, LITMUS^{RT} adds four types of special-purpose virtual character devices. Three of these are for tracing purpose: one for debug data, called the `TRACE()` device; one for overhead sample, called the *Feather-Trace* device; and

one for scheduling events, called the `sched_trace()` device. Tracing is in-depth explained in Section A.3. The fourth device exploits the *control page* mechanism to support low overhead non-preemptive sections.

Since enter and exit from a critical section by disabling interrupt requires two system calls, kernel may incurs in high overhead. Often, this overhead may be higher than the cost to execute the critical section, for example, if a spin-based locking mechanism is used. Thus, the control page enables the kernel and each real-time process to share flags and information (e.g.,, non-preemptive status) without requiring system calls. When a process uses the device, the driver allocates a page of kernel memory (the control page) and maps it into the address space of the process. See [Brandenburg, 2011] for more details.

The user-space interface to LITMUS$^{\mathrm{RT}}$ is finally provided by the *liblitmus* library. It contains all necessary system calls and definitions to interact with the kernel services. The most important tools and their purpose are listed in Table A.2.

Table A.2: List of tools in the *liblitmus* user-space library.

Tool	Purpose
`setsched`	Select active plugin
`showsched`	Print the name of the currently active scheduler
`rt_launch`	Launch an executable as a real-time task provisioned with the worst-case execution time and priod. Mainly used for debugging purpose
`rtspin`	A simple dummy task for emulating purely CPU-bound workloads
`release_ts`	Release the task system. This allows for synchronous task system releases
`measure_syscall`	A simple tool that measures the cost of a system call
`cycles`	Display cycles per time interval
`base_task`	Example real-time task. Can be used as a basis for the development of single-threaded real-time tasks
`base_mt_task`	Example multi-threaded real-time task. Use as a basis for the development of multithreaded real-time tasks

A.3 Tracing with LITMUS^{RT}

There are two tracing mechanisms available in LITMUS^{RT}.

A.3.1 The `TRACE()` macro

The `TRACE()` macro, which is part of the LITMUS^{RT} core infrastructure, is equivalent to the `printk()` function, which allows to export debugging informations from the kernel. The `TRACE()` macro substitutes the `printk()` function since the latter may lead to a system deadlock [Brandenburg, 2011]. In order to avoid this inconvenient, a `TRACE()` device is polled to exports concatenated messages. This solution does not incur in a deadlock but significantly increases the overhead.

A.3.2 The *feather-trace* tracing toolkit

Feather-Trace is a light-weight tracing toolkit for general purpose operating systems. It allows to trace the the system behaviour by collecting performance and state data during the execution for a later offline analysis. Its simplicity and efficiency make it attractive for using it in LITMUS^{RT}.

Feather-Trace provides two components:

- static triggers, which are used to redirect the flow of execution to a user-provided callback function that can take appropriate actions. The negligible overhead is achieved by exploiting the semantic of the unconditional jump assembly instruction (JMP). A trigger is disabled by default and the jump instruction is set to skip over the trigger code. Therefore, the activation is sufficiently possible by giving a null offset to the jump instruction which effectively disable it.

- a wait-free multi-writer, single-reader FIFO buffer, which is used for low-overhead data collection. The writer can execute without waiting on the buffer and without performing expensive copy operations. The reader, typically a non-real-time background process, periodically flushes the buffer to the disk. Since the buffer has a bounded capacity, it may exhaust if the reader is not able to execute. This problem may emerge when the system is close to the full utilization.

Implementation details can be found in: [Brandenburg and H, 2007]

In LITMUS^{RT}, Feather-Trace is used both to record timestamps at various points during the execution of the scheduler and to record the scheduling events (e.g., job releases, preemptions, job completions, etc.). Timestamps before and after an event

allow to reconstruct how long each event took. Scheduling events are recorded in a similar way, but these can be (offline) plotted to obtain a visual depiction of the recorded schedule, to find scheduling errors and also to generated scheduling statistics. Events are exported to user space by two different virtual devices: the `sched_trace()` device and the `feather-trace` device for scheduling events and overhead events respectively.

Appendix B

Using the RUN plugin

The are several steps to follow in order to perform experiments on the RUN plugin. Most of these are related to the LITMUS^{RT} testbed and can be found on the project site. Hence, this appendix focuses on the usage differences between the RUN plugin and the others.

In the following it is assumed:

- a platform running the LITMUS^{RT} kernel built together with our RUN plugin
- `liblitmus` tools and `feather-trace-tools` built and stored in the filesystem

The experiment generation is the most important phase which has to be taken into account. It comes before the experiment execution and it defines the systems parameters, such as the task set, in order to produce significant results. This phase is made offline and can take advantage of the `experiment-scripts`[1] repository which contains a set of scripts whose purpose is to provide a common way for creating, running, parsing, and plotting experiments using LITMUS^{RT}.

Since these scripts are only suitable to the original LITMUS^{RT} plugins, we provide a modified version of the `gen_exps.py` generation script, in particular, which is able to perform the task reduction necessary to use the RUN plugin. The following commands, for example, generate different experiments to be used with the RUN plugin.

```
gen_exps.py -o exps
RUN
cpus = 8
max_util = 4.0,4.5,5.0
utils = bimo-medium
```

[1] https://github.com/hermanjl/experiment-scripts

```
gen_exps.py -o exps
RUN
cpus = 4,8
tasks = 16,24
utils = uni-medium
```

Since the `max_util` parameter is not present in the original script release, we appositely added it to the script to limit the system utilization according to our experiment objectives. The first command generates three experiments for the RUN scheduler, with an utilization cap of `4.0`, `4.5` and `5.0` respectively. Task utilizations are randomly distributed over two intervals according to a bimodal distribution. Consequently, the number of tasks depends on the `max_util` parameter. The second command, instead, combines the `tasks` and `cpus` parameters in order to produce four experiments. In this case, task utilizations are randomly generated to follow an uniform distribution. An execution of the `gen_exps.py` script has to generate three files:

- a `params.py` file which specifies the experiment parameters (i.e., the plugin to be used, the duration of the experiment, the number of cpus, etc.). A file example is the following:

  ```
  {'cpus': 16,
   'duration': 30,
   'max_util': '15.4',
   'scheduler': 'RUN',
   'pre': 'run_config.sh',
   'utils': 'bimo-heavy'}
  ```

  It is worth noting that we added a `pre` parameter whose purpose is to invoke an external script before the start of the experiments. In particular, it allows to clone the reduction tree after the selection of the plugin but before starting any tasks.

- the `sched.py` file contains the specification of the task set. All task parameters are specified in this file. For example it contains:

  ```
  -s 0.95 -S 3 23 50
  -s 0.95 -S 0 159 200
  -s 0.95 -S 3 4 25
  -s 0.95 -S 2 28 50
  -s 0.95 -S 1 70 100
  ```

The `-S 3` parameter indicates that, once online, the task is admitted by the server 3. The first parameter is, instead, the scaling value used to accommodate the scheduling overhead. Finally, other parameters are respectively the WCET and the period of the task.

- the `tree.json` file is the outcome of the reduction. It represents the reduction tree and it must be used during the initialization phase of the plugin, as we explained.

In order to avoid a relative path issue during the initialization phase, we also created a Linux bash script which invokes the cloning tool `run_config` according to the `pre` parameter. The code is reported here:

```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
run_config $DIR/tree.json
```

It first obtains the directory in which the reduction tree file is stored, then, it invokes the `run_config` tool. The `run_config` tool, written in `C` language, performs several system calls in order to clone each tree node into the kernel space, one at a time. In order to simplify the parsing of the json file, we exploited the `cJSON`[2] library.

We also created a repository called `RUNTools` which contains a stand-alone reducing tool. It reduces a given task set allowing to change some parameters such as the packing heuristic, and it produces the reduction tree in the `JSON` file format as well as in the `HTML` graphical version. The tool is publicly available at https://github.com/dcompagn/RUNTools.

The `run_exps.py` has also been modified in order to execute the pre-execution script as explained above.

When it comes to the analysis of the collected traces, all the tools have been developed from scratch using the python language. These tools parse the trace files produced by the `feather-trace-tools` library.

We focused on two tools in particular:

- the `mig_counter.py` performs the scheduling statistics as reported in the results. In detail, it produces statistics for each task as well as the entire task set, such as the total number of jobs, the total number of preemptions and migrations, the total number of missed deadlines, and so on. It also produces the median and the variance for all these values.

- the `overhead_parser.py`, instead, performs statistics obtained from the overhead traces. In the same way, we measured the max, min, avg and variance of

[2] http://sourceforge.net/projects/cjson/

the collected overhead values for both the release and the schedule primitive, as reported in the charts.

Since these scripts produce statistics for a single experiment, we also created additional scripts which coherently assemble the results of each experiments and produce the graphs.

Appendix C

DATE 2014 submission

Putting RUN into practice: implementation and evaluation

Removed for blind review

***Abstract*—The Reduction to UNiprocessor (RUN) scheduling algorithm represents an original approach to multiprocessor scheduling that exhibits the prerogatives of both global and partitioned algorithms, without incurring the respective disadvantages. As an interesting trait, RUN promises to reduce the amount of timing interference across cores, while being at the same time theoretically optimal. However, RUN has also raised some doubts on its practical viability in term of incurred overhead and required kernel support. Surprisingly, to the best of our knowledge, no practical implementation and empirical evaluation of it have been presented yet. 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 and supporting much higher schedulable utilization than its partitioned and global counterparts.**

I. Introduction

Compelling energy, performance and availability considerations push towards the migration from relatively simple single-processor systems to more complex multiprocessors platforms. The latter in fact provide the necessary computational power without incurring thermal and energy drawbacks. The deployment of hard and soft real-time systems on top of multiprocessor systems requires novel scheduling policies and analysis techniques to guarantee that their real-time requirements can always be met. Despite the increasing research interest and recent advances and results, 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 rely on a static mapping that assigns tasks to processors, so that each task can be only scheduled on the processor it has been assigned to. Global approaches instead do not assume any fixed task-to-processor allocation and tasks or jobs are allowed to *migrate* from one processor to another, upon task preemptions and suspensions. None of these methods, 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 is substantially equivalent to the NP-hard bin-packing problem. Global algorithms, on the other hand, while being naturally work-conserving and thus able to guarantee higher utilization, bring in the additional overhead incurred 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 and general solution to the multiprocessor scheduling problem comes out in favour of 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.

Within this line of thought, the *Reduction to UNiprocessor* (RUN) algorithm, first introduced in [5], represents an original and interesting approach. When applied to homogeneous multiprocessor systems, comprising *m* processors, RUN employs particular scheduling abstractions such as *primal* and *dual servers* to build an off-line data structure (*reduction tree*), which is used to reduce the scheduling problem to *m* single processor schedules and to guide scheduling decisions at run time. Those scheduling decisions are designed to minimize the number of migrations and thus the incurred overhead. Moreover, 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 widely appreciated for the elegance of the proposed solution, RUN has also given rise to concerns on its practical implementation, mainly due to the building process and the data representation of the reduction tree, and the overhead expected from frequently updating it. Those doubts can only be confirmed or refuted by an empirical evaluation of the algorithm. Quite surprisingly, to the best of our knowledge, no empirical evaluation of RUN or comparison with other algorithms have been performed yet. In this paper, we provide the first solid implementation of RUN with the twofold objective 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 effort we use LITMUS$^{\mathrm{RT}}$ [6], [7], a real-time extension to Linux published by 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 recall 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 briefly 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.

II. Related Work

Most work on multiprocessor scheduling has focused on partitioned and global approaches and 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. Moreover, adding the task allocation problem to system design, and solving it by heuristics, is not an industrial-quality practice in any resource-constrained development. 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 have been proved for the family of *proportional 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 realizations of those hybrid approaches are likely to require much more complex implementations, on the score of the fact that they do retain characteristics of both global and partitioned approaches. Therefore, when reasoning on the performance of these algorithms we must account for 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 large collection of schedulers have 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.

When it comes to RUN, however, no solid evaluation has been conducted yet. A tentative implementation, still on top of LITMUS$^{\mathrm{RT}}$, has been sketched in [15]. The extremely brief description therein reported, however, seems to diverge from the strict formulation of RUN in that it simply casts the reduction tree to static table-driven scheduling. In our implementation, instead, we stick to the original formulation of RUN as we deem the table-driven approach too rigid and incapable of accommodating release jitter.

III. The RUN Scheduler

Before the advent of RUN, optimality for multiprocessor scheduling was proved for the family of algorithms based on *proportional fairness*, whose progenitor was PFair [9], [10]. RUN aimed 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.

For reasons of space limitations, in the following we only outline our implementation of RUN on top of LITMUS$^{\mathrm{RT}}$. Before presenting its main characteristics, 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 $J.c$ and a deadline $J.d$, executing on top m homogeneous processors.

Aiming at representing possibly non-periodic execution, RUN branches off from the canonical sporadic task model and introduces the *fixed-rate* task model. Under this model tasks are characterized by a constant execution rate $\mu \leq 1$ and an infinite set of deadlines D. 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 given in [5] ambiguously refers 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], builds on the observation that solving the problem of scheduling a given task set $\mathcal{T} \ni \{\tau_i\}_{i\in\{1,n\}}$ is equivalent to that of scheduling the *dual* task set $\mathcal{T}^*$, where the latter is comprised 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.

By alternately packing tasks (servers) and reformulating the scheduling problem into its dual, the number of processors is progressively reduced and eventually reaches one [5], thus reducing the initial problem to a uniprocessor scheduling problem. 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.

Figure 1 shows an example of reduction and the resulting reduction tree. Reduction proceeds in a bottom-up fashion. As a first step, tasks are packed into level-0 servers under the condition that the cumulative utilization cannot exceed 1: for example τ_1 and τ_2 are packed together into S_1 (and so do τ_5 and τ_6 in S_4), while 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. The next step consists in switching to the dual representation of the problem, which originates the respective set of dual servers S_i^*. This sequence of steps is then repeated until we get a set of servers S_5^*, S_6^*, S_7^* with a total utilization 1, thus feasible on a single processor. By packing the last level of dual servers into a *unit* server S_8 we set down the root of the reduction tree, whose leaves regroup the original task set. The assumption on the full system utilization guarantees the termination of the reduction.

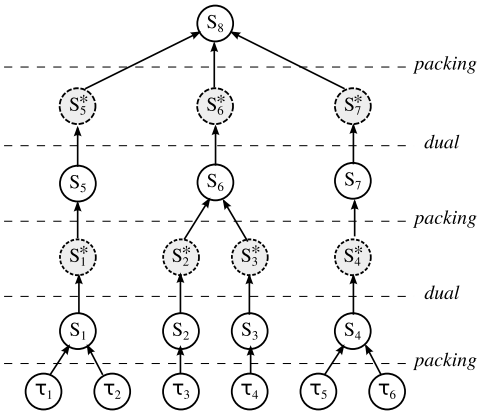


Fig. 1. RUN reduction.

At run time, scheduling is performed according to the information in the reduction tree, which is dynamically updated. 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 at each instant t. The correctness of the relationship between primal and dual schedule is guaranteed by RUN assumptions on full utilization and fixed-rate requirements.

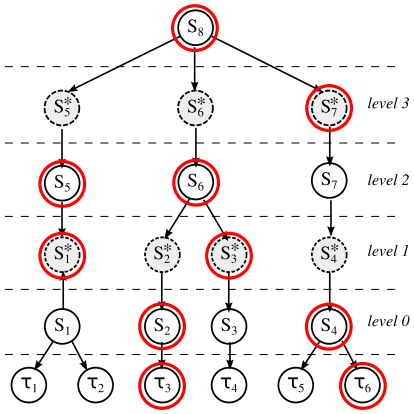


Fig. 2. RUN scheduling.

Figure 2 shows a possible task selection as determined by the state at time t of the same reduction tree reported in Figure 1. In this case, tasks τ_3 and τ_6 have been selected for execution. In contrast with G-EDF, RUN is prevented from making greedy scheduling decisions by the proportional 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.

RUN has been classified by its authors under the umbrella of semi-partitioned algorithms. We contend however that RUN is not a semi-partitioned approach in the common acceptation of the term, and falls into a separate category. RUN is a hybrid approach that may behave more or less similarly to a global o partitioned algorithm: the packing step which pairs RUN with partitioned approaches is rather particular as tasks are not associated to processors but to servers, which in principle are not pinned to any particular processor.

On the one hand, RUN is global in the sense that scheduling decisions are taken globally and the packing is much less critical to the final performance of the algorithm as compared to, for example, P-EDF: it may influence the shape of the reduction tree, but it cannot compromise the validity of a schedule (when P-EDF packing does). On the other hand, with favourable task sets, the reduction tree may produce exactly the same behaviour as P-EDF. Being partitioned or global depending on the task set is a very valuable property 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 2012.3 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 available to the 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 P-Fair 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, clear similarities (i.e., global release events) suggested that the LITMUS$^{\mathrm{RT}}$ implementation of G-EDF was a good base for the implementation of RUN.

C. *The RUN plug-in*

Although the description of RUN in [5], [17] leaves ample room for design decisions and optimizations, our effort was mainly oriented toward a simple and direct implementation of it[1]. Accordingly, we focused on providing a convenient data structure for the reduction tree and ensured it would be

[1]Our plug-in is publicly available at *Removed for blind review*.

consistently updated at run time with fresh deadline and budget information [5].

In this first implementation attempt, no particular attention has been paid to the way tasks and servers are packed together in the reduction tree, and a simple *worst-fit* policy has been followed. Different policies may lead to different reduction trees, which in turn may result in different preemption and migration patterns at run time.

We had to figure how to live 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 has been 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 is ready. Within the scheduling domain of a level-0 EDF server, if a job J completes its execution before $J.r+J.c$ then either we execute the next EDF job in the server, if any, or just do nothing, thus allowing the execution of native tasks. Advancing in the local schedule, in fact, does not impair the overall schedule. When it comes to (ii), instead of further increasing the task population, we preferred to devolve the available slack time to a buffer for run-time kernel overheads, by assigning proportional shares of system slack to each level-0 server.

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 [18], 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 (i.e., viability) and exhibit actual kernel overheads that may not always be evident. The proposed implementation of RUN on top of LITMUS$^{\mathrm{RT}}$ 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 studying the scheduling algorithm with respect to the interference it causes to the system rather than finding its empirical utilization caps. For this reason, we set on assessing the kernel overhead in terms of both the cost of its primitives and the number of preemptions/migrations incurred. Accordingly, we did not evaluate RUN against other optimal algorithms: we wanted, instead, to compare it against P-EDF[2] and G-EDF (both included in the LITMUS$^{\mathrm{RT}}$ 2012.3 release) as representatives of global and partitioned algorithms. 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 guarantee a fair experimental process, 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 utilizations as we expect most important variations to occur then. Task utilizations, instead, 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%. Moreover, tasks were constrained to exhibit harmonic periods drawn from a uniform distribution in [25ms; 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.

Since we expected kernel overheads to also vary on the number of cores available, we performed three sets of experiments targeting respectively 4, 8 and 16 cores. Experiments were run on a dual-processor Intel Xeon E5-2670 system where each processor includes 8 cores running at 2.6 GHz and 20 MB of cache. Hyper-threading and power management functions were intentionally disabled. We are aware that the cost of preemptions and migrations may be largely determined by cache-related interference [8], [19]. However, in our setting, cache effects are negligible because tasks feature extremely reduced working sets (i.e., tasks simply execute an empty loop). Hence the reported overheads are not representative of cache-related effects: we defer the quantification of those overheads to future work.

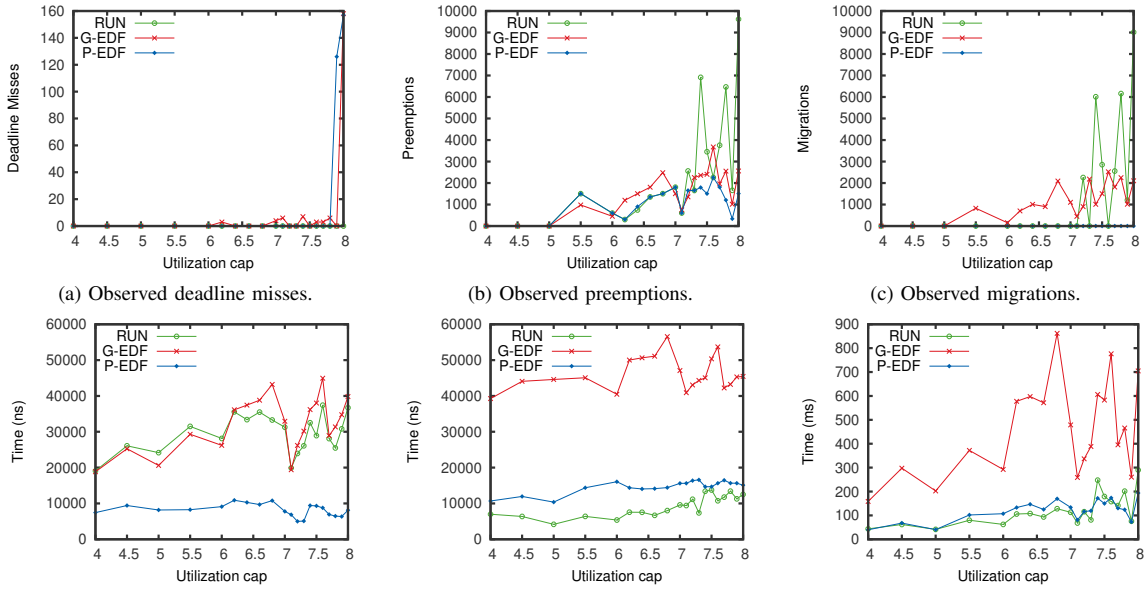
B. Results

Owing to space constraints, we do not present here the graphs for the 4-cores experiments; we do so with no loss of information as they just exhibit the same trend observed for 8 and 16 cores. It should be noted that single task utilizations includes a 5% slack, to allow for kernel and tracing overheads. Consequently also the reported overall system utilization is just nominal.

Figure 3 reports the results observed on 8 cores (located on the same processor). As shown in Figure 3a, RUN is capable of sustaining extremely high workloads, up to a theoretical 100% peak. P-EDF is penalized by higher utilizations as it cannot guarantee good solutions to the packing problem (cured instead by dual representations in RUN). Conversely, G-EDF already produces invalid schedules at medium utilization levels. We conjecture that the kernel overhead incurred by RUN is sufficiently low to fit in the provided 5% slack. When it comes to kernel interference, the number of preemptions incurred by RUN (Figure 3b) is quite similar to those observed for P-EDF. 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 less preemptions than RUN; however, almost all such preemptions result in a job migration, as shown by comparing Figure 3b and 3c. On the contrary, RUN only incurs some migrations at extremely high utilizations and behaves worse than G-EDF only in those cases where G-EDF produces an invalid schedule.

The plain overhead incurred by each scheduling algorithm,

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



(a) Observed deadline misses. (b) Observed preemptions. (c) Observed migrations.

(d) Average job release overhead. (e) Average dispatching overhead. (f) Cumulative kernel overhead.

Fig. 3. Experimental results on 8 cores under increasing system utilization.

instead, is reported in Figure 3d and 3e, showing the average execution time observed for the release and schedule primitives. It is worth noting that the cost of updating the reduction tree (quite small in practice) is included in both primitives. As expected, the use of local (per partition) data structures causes P-EDF to outperform the other algorithms relatively to the cost of a job release. The fact that the cost of a job release in RUN is quite modest in comparison to G-EDF depends on the optimization we applied when handling simultaneously multiple job releases. 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 but uses a separate queue for each level-0 server. The cost of a schedule in RUN is thus similar to that of P-EDF. The effects of this favourable condition is an extremely low cumulative overhead, as shown in Figure 3f.

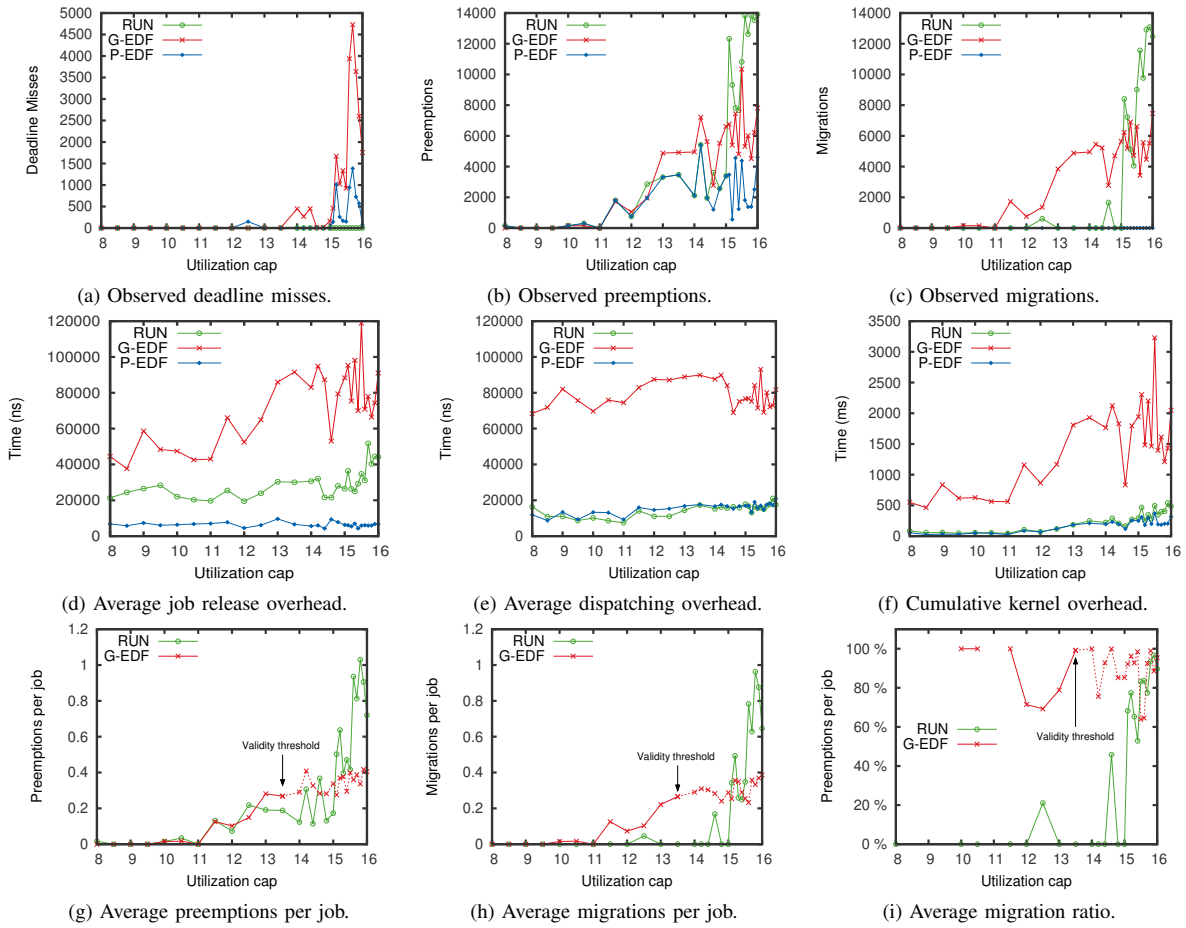
Similar experiments were conducted on 16 cores, whose results are reported in Figure 4. In this case, we should consider that experiments were actually executed on two 8-core processors and thus kernel overheads also include inter-processor effects. This fact, however, does not invalidate the results. As in the 8-core experiments, P-EDF and G-EDF cannot always guarantee all jobs to meet their deadline (Figure 4a): again, our comparison focuses exclusively on valid schedules. The observations on preemptions and migrations, reported in Figure 4b and 4c respectively, confirm the trend from the 8-core experiments. From Figure 4b we observe that RUN generally incurs the same number of preemptions as P-EDF and always less than G-EDF. As the only relevant exception, at utilization 12.5 RUN seems to suffer more preemptions than the other algorithms. However, we also note that in that case P-EDF incurs some deadline misses (see Figure 4a) since it was unable to find a perfect packing. In this case RUN did not find a perfect packing either, and was forced to resort to a 2-level reduction tree which is a potential source of preemptions. As for migrations, instead, RUN always outperforms G-EDF (Figure 4c).

The overhead incurred by the scheduling primitives is reported in Figures 4d-4f. The three diagrams confirm the trend observed in the 8-core experiments: the overall cumulative overhead suffered by RUN is quite low and almost assimilable to P-EDF. Moreover, comparing this series of diagrams with their 8-core counterparts, we also notice that both P-EDF and RUN scale exceptionally well with the number of cores whereas the overhead suffered by G-EDF explodes.

Figures 4g and 4h show the average number of preemptions and migrations per job under G-EDF and RUN. These diagrams mirror the cumulative behaviour reported in the previous set of figures, but break it per job. A more interesting insight can be drawn from Figure 4i, reporting the ratio between migrations and preemptions. For low and medium utilization levels RUN causes less migrations than G-EDF, which instead shows a high probability of migration. For higher utilization levels also the migration ratio in RUN increases: this could be seen as the price that has to be paid to guarantee task set feasibility.

V. Conclusion

We presented in this paper the first solid implementation of the RUN multiprocessor scheduling algorithm on top of the LITMUS$^{\mathrm{RT}}$ test bed. 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 costs and practical viability. Our experimental results suggest that depending on the underlying implementation RUN may exhibit quite moderate kernel overhead, almost similar to that of a partitioned algorithm, such as P-EDF. In addition, our experiments also confirmed that RUN minimizes the number of migrations with respect 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 confirmed that the number of preemptions and migrations in RUN heavily depends on



(a) Observed deadline misses. (b) Observed preemptions. (c) Observed migrations.

(d) Average job release overhead. (e) Average dispatching overhead. (f) Cumulative kernel overhead.

(g) Average preemptions per job. (h) Average migrations per job. (i) Average migration ratio.

Fig. 4. Experimental results on 16 cores under increasing system utilization.

the way the off-line reduction tree is built, including but not limited to, the packing heuristic adopted. We plan to further investigate practical means to build *good* reduction trees that can further reduce the number of preemptions and migrations. As an orthogonal line of research, we are studying whether and how RUN could accommodate *real* sporadic tasks, whose job release time is not known beforehand.

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, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011.

[6] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *IEEE 27th Real-Time Systems Symposium (RTSS)*, 2006, pp. 111–126.

[7] LITMUS^{RT}, "The Linux Testbed for Multiprocessor Scheduling in Real-Time Systems," 2013, 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^{RT}," in *WiP Session RTSS*, 2012.

[16] G. Nelissen, V. Berten, V. Nélis, J. Goossens, and D. Milojevic, "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] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

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

Bibliography

Altmeyer, S. and C. Burguiere

2009 "A new notion of useful cache block to improve the bounds of cache-related preemption delay", in *ECRTS*. (Cited on p. 40.)

Anderson, J. and A. Srinivasan

2001 "Mixed Pfair/ERfair scheduling of asynchronous periodic tasks", in *ECRTS*. (Cited on p. 7.)

Andersson, B. and K. Bletsas

2008 "Sporadic Multiprocessor Scheduling with Few Preemptions", in *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pp. 243–252, DOI: 10.1109/ECRTS.2008.9. (Cited on p. 9.)

Andersson, Bjorn, Sanjoy Baruah, and Jan Jonsson

2001 "Static-Priority Scheduling on Multiprocessors", in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, IEEE Computer Society, Washington, DC, USA, pp. 93–, ISBN: 0-7695-1420-0, http://dl.acm.org/citation.cfm?id=882482.883823. (Cited on p. 6.)

Baruah, S. K., N. K. Cohen, C. G. Plaxton, and D. A. Varvel

1996 "Proportionate progress: A notion of fairness in resource allocation", *Algorithmica*, 15, pp. 600–625. (Cited on p. 7.)

Baruah, Sanjoy K., Johannes E. Gehrke, and C. Greg Plaxton

1995 "Fast Scheduling of Periodic Tasks on Multiple Resources", in *Proceedings of the 9th International Parallel Processing Symposium*, pp. 280–288. (Cited on p. 7.)

Bastoni, Andrea, Bjorn Brandenburg, and James Anderson

2011 "Is Semi-Partitioned Scheduling Practical?", in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*. (Cited on pp. 6, 10, 21, 23, 33, 40.)

Bertogna, Marko

2008 *Real-Time Scheduling Analysis for Multiprocessor Platforms*, PhD thesis, Scuola Superiore Sant'Anna, Pisa. (Cited on p. 1.)

Bletsas, K. and B. Andersson

2009 "Preemption-Light Multiprocessor Scheduling of Sporadic Tasks with High Utilisation Bound", in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 447–456, DOI: 10.1109/RTSS.2009.16. (Cited on p. 7.)

Brandenburg, Björn B.

2011 *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*, PhD thesis, The University of North Carolina at Chapel Hill. (Cited on pp. 21, 60, 61.)

Brandenburg, Bjorn B. and James H

2007 "Feather-trace: A light-weight event tracing toolkit", in *In Proceedings of the Third International Workshop on Operating Systems Platformsfor Embedded Real-Time Applications (OSPERT'07*, pp. 61–70. (Cited on p. 61.)

Burns, A. and A.J. Wellings

2009 *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, International Computer Science Series, Addison-Wesley, ISBN: 9780321417459, http://books.google.it/books?id=XO9dPgAACAAJ. (Cited on p. 1.)

Calandrino, J., J. Anderson, and D. Baumberger

2007 "A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms", in *Proceedings of the 19th Euromicro Conference on Real-Time Systems*. (Cited on p. 8.)

Calandrino, John, Hennadiy Leontyev, Aaron Block, UmaMaheswari Devi, and James Anderson

2006 "LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers", in *IEEE 27th Real-Time Systems Symposium (RTSS)*, pp. 111–126. (Cited on p. 10.)

Chishiro, Hiroyuki, James Anderson, and Nobuyuki Yamasaki

2012 "An Evaluation of the RUN Algorithm in LITMUS^{RT}", in *WiP Session RTSS*. (Cited on p. 23.)

Cormen, Thomas H., Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson

2001 *Introduction to Algorithms*, 2nd, McGraw-Hill Higher Education, ISBN: 0070131511. (Cited on pp. 25, 28.)

Davis, Robert I. and Alan Burns

2011 "A survey of hard real-time scheduling for multiprocessor systems", *ACM Comput. Surv.*, 43, 4. (Cited on pp. 1, 6.)

Gai, P., M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca

2003 "A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform", in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pp. 189–198, DOI: 10.1109/RTTAS.2003.1203051. (Cited on p. 7.)

Horn, W. A.

1974 "Some simple scheduling algorithms", *Naval Research Logistics Quarterly*, 21, 1, pp. 177–185, ISSN: 1931-9193, DOI: 10.1002/nav.3800210113, http://dx.doi.org/10.1002/nav.3800210113. (Cited on p. 5.)

Kato, S., N. Yamasaki, and Y. Ishikawa

2009a "Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors", in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pp. 249–258, DOI: 10.1109/ECRTS.2009.22. (Cited on p. 9.)

Kato, Shinpei, Nobuyuki Yamasaki, and Yutaka Ishikawa

2009b "Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors", in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*. (Cited on p. 7.)

Levin, Greg, Caitlin Sadowski, Ian Pye, and Scott Brandt

2009 "SnS: a simple model for understanding optimal hard real-time multiprocessor scheduling", *Univ. of California, Tech. Rep. UCSCSOE-11-09*. (Cited on p. 11.)

LITMUS^{RT}

2013 *The Linux Testbed for Multiprocessor Scheduling in Real-Time Systems*, http://www.litmus-rt.org/. (Cited on pp. 10, 17.)

Liu, Jane W. S. W.

2000 *Real-Time Systems*, 1st, Prentice Hall PTR, Upper Saddle River, NJ, USA, ISBN: 0130996513. (Cited on p. 2.)

McKenney, P.

2005 "Shrinking slices: Looking at real time for Linux, PowerPC, and Cell", https://www.ibm.com/developerworks/power/library/pa-nl14-directions/. (Cited on p. 17.)

Regnier, P.

2012 *Optimal multiprocessor real-time scheduling via reduction to uniprocessor*, PhD thesis, UFBS. (Cited on pp. 16, 23, 41.)

Regnier, P., G. Lima, E. Massa, G. Levin, and S. Brandt

2011a "RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor", in *IEEE 32nd Real-Time Systems Symposium (RTSS)*. (Cited on pp. 5, 10, 12, 13, 16, 23, 24, 26, 31, 33, 35, 37, 41.)

Regnier, Paul, George Lima, and Ernesto Massa

2011b "An Optimal Real-Time Scheduling Approach: From Multiprocessor to Uniprocessor", *CoRR*, abs/1104.3523. (Cited on p. 30.)

Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne

2008 *Operating System Concepts*, 8th, Wiley Publishing, ISBN: 0470128720. (Cited on p. 41.)

Torvalds, L.

1994 "Linux kernel implementation", in *Proceedings of the AUUG94 Conference: Open systems. Looking into the future: 6–9 September 1994, World Congress Centre, Melbourne, Australia*, ed. by Anonymous, AUUG Inc, Kensington, NSW, Australia, pp. 9–14, ISBN: 0-646-20108-5. (Cited on pp. 16, 34.)