Real-Time Syst
DOI 10.1007/s11241-012-9165-x

# Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach

**Paul Regnier · George Lima · Ernesto Massa · Greg Levin · Scott Brandt**

**Abstract** Optimal multiprocessor real-time schedulers incur significant overhead for preemptions and migrations. We present RUN, an efficient scheduler that reduces the multiprocessor problem to a series of uniprocessor problems. RUN significantly outperforms existing optimal algorithms with an upper bound of $O(\log m)$ average preemptions per job on $m$ processors (fewer than 3 per job in all of our simulated task sets) and reduces to Partitioned EDF whenever a proper partitioning is found.

**Keywords** Real-time · Multiprocessor · Scheduling · Server

A preliminary version of this work has been published in the Proceedings of the 32nd IEEE Real-Time Systems Symposium, 2011, pages 104–115 (Regnier, Lima, Massa, Levin and Brandt 2011) held in Vienna, Austria. It received the Best Paper Award.

P. Regnier (✉) · G. Lima
Federal University of Bahia, Salvador, Brazil
e-mail: pregnier@ufba.br

G. Lima
e-mail: gmlima@ufba.br

E. Massa
State University of Bahia, Salvador, Brazil
e-mail: ernestomassa@ufba.br

G. Levin · S. Brandt
University of California, Santa Cruz, USA

G. Levin
e-mail: glevin@soe.ucsc.edu

S. Brandt
e-mail: sbrandt@soe.ucsc.edu

Published online: 10 November 2012

# 1 Introduction

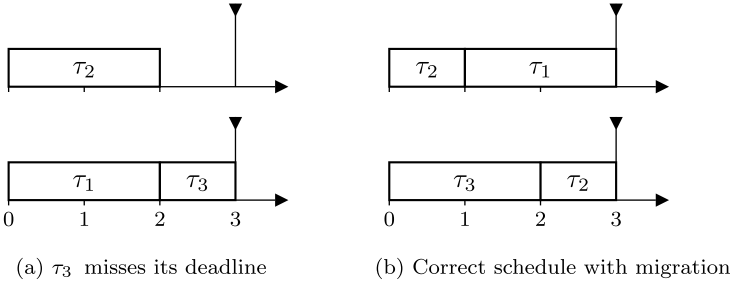
## 1.1 Motivation

As multiprocessor systems become common-place, interest in multiprocessor scheduling algorithms has increased. However, optimal multiprocessor scheduling in a real-time environment remains a challenging problem. Several solutions have recently been presented, most based on periodic-preemptive-independent tasks with implicit deadlines (PPIID). We address a slight generalization of this task model, with the goal of finding efficient schedules wherein all task deadlines are met.

Multiprocessor scheduling is often accomplished via *partitioned* approaches in which tasks are statically assigned to processors, guaranteeing only 50 % utilization in the worst case (Koren et al. 1998). *Global* approaches can achieve full utilization by migrating tasks between processors, at the cost of increased runtime overhead. For example, consider a two-processor system with three tasks, $\tau_1$, $\tau_2$ and $\tau_3$, each requiring 2 units of work every 3 time units. If two tasks are scheduled on the two processors and run to completion, the third task cannot complete on time (see Fig. 1(a)). However, if tasks may migrate, then all three may be completed in the time available (Fig. 1(b)). This is a simple example of McNaughton's wrap-around algorithm (McNaughton 1959), which works whenever all jobs have the same deadline.

We are interested in *optimal* scheduling algorithms, which always find a valid schedule whenever one exists, up to 100 % processor utilization. Several optimal algorithms have been developed (Andersson and Tovar 2006; Baruah et al. 1996; Cho et al. 2006; Levin et al. 2010; Zhu et al. 2003, 2011), all relying on *proportionate fairness*, either explicitly or in some modified form. This over-constraining technique, first introduced by Baruah et al. (1993), requires that tasks' executions match their ideal long-term execution rates, not just at their deadlines, but at numerous points in between. Most of these algorithms rely on the simplicity of McNaughton's approach, and enforce deadline equality by proportionally subdividing workloads and imposing the deadlines of each task on all other tasks (Levin et al. 2010). This causes many tasks to execute between every pair of consecutive system deadlines, leading to excessive context switching and migration overhead.

In this article, we present RUN (Reduction to UNiprocessor), the first optimal multiprocessor scheduling algorithm not based on proportionate fairness which achieves a significantly lower overhead than previous approaches.



$\tau_2$
$\tau_1$
$\tau_3$
0
1
2
3
(a) $\tau_3$ misses its deadline
$\tau_2$
$\tau_1$
$\tau_3$
$\tau_2$
0
1
2
3
(b) Correct schedule with migration

**Fig. 1** (**a**) Partitioned and (**b**) Global scheduling approaches

## 1.2 Approach

We consider a real-time platform comprised of $m \geq 1$ identical processors, each executing tasks at a uniform rate. We focus on *global* scheduling, wherein all processors and tasks are scheduled by a single dispatcher.

We define a real-time task as an infinite sequence of jobs. Each job is a portion of work to be executed between its release time $r$ and deadline $d$. We assume that tasks are independent and fully preemptable with free migration between processors. Also, we assume that jobs have implicit deadlines, i.e., the deadline of a task's job is the release time of its next job. However, we do not assume that tasks are periodic. Instead, we assume that each task has a fixed *rate*, which is the fraction of a processor that it utilizes. Since a task must be schedulable on a single processor, its rate is necessarily no more than 1. If a task has rate $\rho$, then a job of this task with release time $r$ and deadline $d$ will require $\rho(d-r)$ execution time. The PPIID model for periodic tasks is a special case of this formulation.

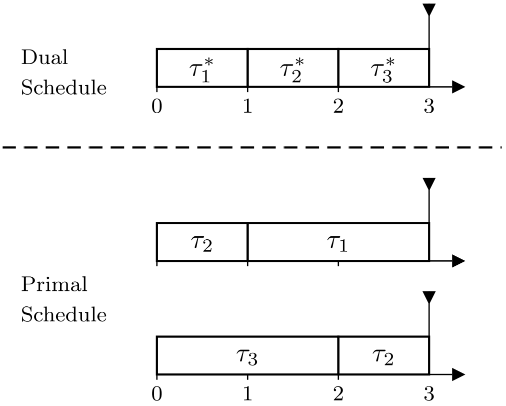
RUN produces a valid multiprocessor schedule for a set of fixed-rate tasks by (1) reducing the real-time multiprocessor scheduling problem to an equivalent set of easily solved uniprocessor scheduling problems through two operations: DUALand PACK, (2) solving these problems with well-known techniques, and (3) transforming the solutions back into a multiprocessor schedule.

In the 3-task example of Fig. 1, RUN creates a "dual" task set $\{\tau_1^*, \tau_2^*, \tau_3^*\}$ where each dual task $\tau_i^*$ has the same deadline as $\tau_i$ and a complementary workload of $3-2=1$. The dual $\tau_i^*$ of *primal* task $\tau_i$ represents $\tau_i$'s idle time. Because each dual task has a rate equal to $1/3$, all three may be scheduled on a single virtual processor using EDF (see Fig. 2).

From this virtual dual schedule, we may derive a schedule for our primal system $\{\tau_1, \tau_2, \tau_3\}$. This is done using *Dual Scheduling Equivalence* (DSE), which will be detailed in Sect. 4.1. Briefly, since $\tau_i^*$ represents the idle time of $\tau_i$, we idle $\tau_i$ in the primal system precisely when $\tau_i^*$ is executing in the dual system, and execute $\tau_i$ when $\tau_i^*$ is idling. The resultant primal schedule is also shown in Fig. 2.

The dual transformation in this example is beneficial because it reduces the number of processors in the system. However, for a primal system with many low (less

**Fig. 2** Scheduling equivalence of $\tau_1, \tau_2, \tau_3$ on two processors and $\tau_1^*, \tau_2^*$ $\tau_3^*$ on one processor



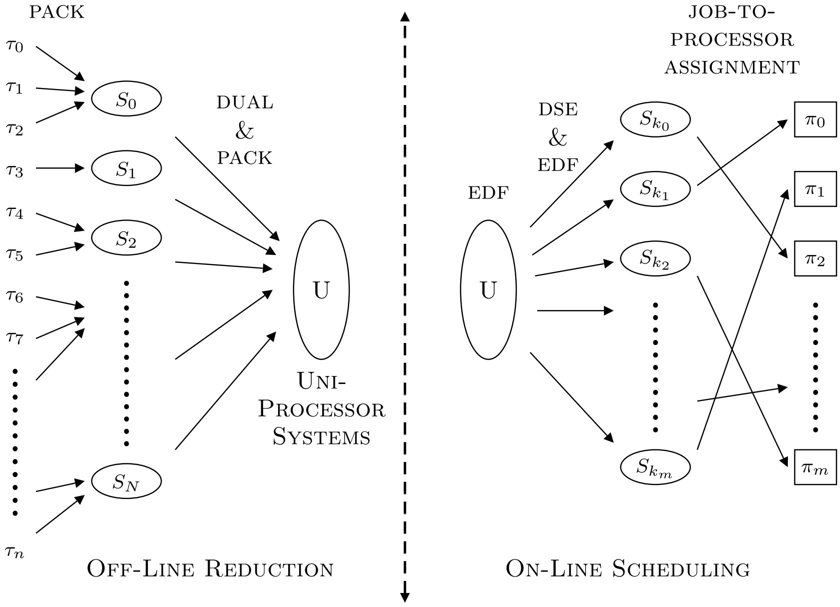


**Fig. 3** The RUN off-line reduction combines the PACK and DUAL operations. The RUN on-line scheduling uses the DSE rule, the EDF algorithm for servers, and a job-to-processor assignment

than 1/2) rate tasks, the dual would consist instead of high rate tasks, and would require *more* processors. To overcome this difficulty, we first aggregate the low rate tasks into fewer high rate tasks, or *servers*, which will act as proxies for their constituent clients. This is illustrated in Fig. 3 by the off-line reduction process. Taking the dual of these high rate servers will result in a system requiring fewer processors than the original.

This, then, is the general approach of RUN: through a sequence of such reduction operations, we iteratively transform a multiprocessor system into one or more uniprocessor systems, denoted $U$ in Fig. 3. This creates a hierarchy of virtual systems, and is done off-line before execution begins. All on-line scheduling decisions come from the application of Earliest Deadline First (EDF) to the virtual uniprocessor systems $U$. Because a dual task and its primal may not execute at the same time, each schedule in the resulting hierarchy constrains the tasks that may execute in the next level down, starting with the uniprocessor schedules and working downward to determine the set of real tasks to execute. Finally, the job-to-processor assignment step allows for the generation of a valid multiprocessor schedule for the original system. This process is summarized in Fig. 3.

### 1.3 Contribution

Our first contributions are the theoretical building blocks of *Dual Scheduling Equivalence* (DSE) and *Deadline Sharing*. DSE uses the dual system to derive feasible schedules, and elaborates on the ideas of duality introduced in Levin et al. (2009).

Deadline sharing requires that a fair share of processor time is allocated to specific subsets (the clients of servers) which share the same deadline instants.

From these, we derive our primary contribution, the RUN scheduling algorithm for periodic and fixed-rate real-time tasks on a system of identical multiprocessors. RUN presents the following advantages:

- Through a sequence of off-line reduction operations, RUN reduces the multiprocessor scheduling problem to a sequence of simpler uniprocessor problems. These are then optimally scheduled on-line by the familiar EDF algorithm.
- RUN significantly outperforms existing optimal algorithms in terms of preemptions. Run has a theoretical upper bound of $O(\log m)$ average preemptions per job on $m$ processors, and an observed average of less than 3 preemptions per job in all of our simulations.
- RUN reduces naturally to Partitioned EDF whenever a proper partitioning can be found.

### 1.4 Structure of this article

The remainder of this article is organized as follows: Sect. 2 precisely describes our system model and notation (summarized in Table 1). In Sect. 3, we define the server abstraction, which is used to aggregate low rate tasks into fewer high rate tasks. These are used in Sect. 4, along with the DUAL and PACK operations, to construct the off-line reduction procedure for transforming a multiprocessor system into a series of uniprocessor systems. Section 5 describes RUN's on-line scheduling procedure, and proves the correctness and optimality of RUN for fixed-rate tasks with implicit deadlines. Section 6 proves various bounds on RUN's performance, including a low theoretical upper bound on average preemptions per job. It also presents the results of extensive simulations and comparisons of RUN with other optimal scheduling algorithms. Section 7 briefly surveys related work, and Sect. 8 presents our conclusions.

**Table 1** Summary of notation

| | |
|---|---|
| $\tau$; $S$; $J$ | Fixed-rate task; Server; Job |
| $\tau : (\rho, \mathrm{R})$ | Task with rate $\rho$, and release times R |
| $J.r$; $J.c$; $J.d$ | Release time; Execution time; Deadline of $J$ |
| $e(J, t)$ | Work remaining for job $J$ at time $t$ |
| $\mathcal{T}$; $\varGamma$ | Set of tasks; Set of servers |
| $\rho(\tau)$; $\rho(\varGamma)$ | Rate of task $\tau$; Rate of server set $\varGamma$ |
| $\mathrm{R}(\tau)$ | Set of release instants of task $\tau$ |
| $\varSigma$ | Schedule of a set of tasks or servers |
| $e(J_S, t)$ | Budget of server $S$ at time $t$ |
| $\mathrm{cli}(S)$ | Set of client servers (tasks) of $S$ |
| $\mathrm{ser}(\varGamma)$ | Server of the set of tasks (servers) $\varGamma$ |
| $\tau^*$; $\varphi(\tau)$ | Dual task of task $\tau$; DUAL operator |
| $\pi_A[\varGamma]$ | Partition of $\varGamma$ by packing algorithm $A$ |
| $\sigma(S)$ | Server of $S$ given by PACK operation |
| $\psi = \varphi \circ \sigma$ | REDUCE operator |

## 2 System model and notation

### 2.1 Fixed-rate tasks

We consider a system of $n$ independent real-time tasks, each representing an infinite sequence of jobs.

**Definition 1** (Job) A real-time *job* $J$, or simply *job*, is a finite sequence of instructions with a release instant $J.r$, a worst-case execution time (WCET) $J.c$, and a deadline $J.d$.

In a real-time environment, the successful scheduling of a job $J$ requires that it only executes between its release instant and its deadline, and that it executes for exactly $J.c$ time during this interval.

In order to represent potentially non-periodic execution requirements, as we will have with servers in Sect. 3, we define tasks in terms of rates instead of workloads. Since we require that tasks be able to execute on a single processor, we do not allow rates larger than one.

**Definition 2** (Fixed-rate task) Let $\rho \leq 1$ be a positive real number and $\mathrm{R}$ a countably infinite set of non-negative integers which includes zero. The *fixed-rate task* $\tau$ with rate $\rho$ and release times $\mathrm{R}$, denoted $\tau{:}(\rho, \mathrm{R})$, releases an infinite sequence of jobs satisfying the following properties:

1. A job $J$ of $\tau$ is released at time $t$ if and only if $t$ is in $\mathrm{R}$
2. A job $J$ released at time $J.r$ has deadline $J.d = \min\{t \in \mathrm{R}, t > J.r\}$
3. The execution time $J.c$ of job $J$ equals $\rho(J.d - J.r)$

Given a fixed-rate task $\tau$, we denote its rate with $\rho(\tau)$ and its release time set with $\mathrm{R}(\tau)$. As all tasks in this article are fixed-rate tasks, we shall henceforth simply refer to these as "tasks". As implied by item (2) of Definition 2, we assume the implicit deadline model, i.e., that the deadline of $\tau$'s current job is equal to the release time of its next job. Consequently, $\mathrm{R}(\tau) \setminus \{0\}$ is the set of $\tau$'s deadlines, and tasks have exactly one active job at any time. Figure 4 illustrates three jobs from a fixed-rate task $\tau$ with rate $\rho(\tau) = 1/2$.

Periodic tasks are just a special case of fixed-rate tasks. Given a periodic task $\tau$ with initial release time $t = 0$, period $T$ and execution time $C$, this may be represented as a fixed rate task where $\rho(\tau) = C/T$ and $\mathrm{R}(\tau) = T\mathbb{N} = \{kT, k \in \mathbb{N}\}$.

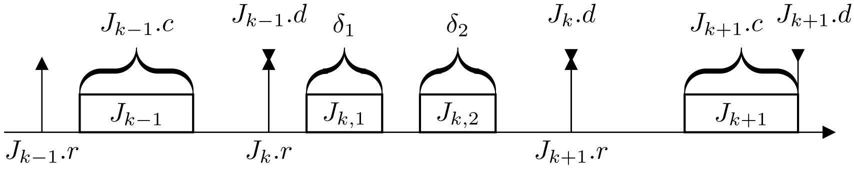


**Fig. 4** Schedule example of jobs $J_{k-1}$, $J_k$ and $J_{k+1}$ of a fixed-rate task $\tau$ where $\rho(\tau) = 1/2$ and $\delta_1 + \delta_2 = J_k.c$

**Definition 3** (Accumulated rate) Let $\mathcal{T}$ be a set of fixed-rate tasks. We say that $\mathcal{T}$ has an *accumulated rate* equal to the sum of the rates of the tasks in $\mathcal{T}$, and denote this by $\rho(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} \rho(\tau)$.

### 2.2 Global scheduling

Jobs are enqueued in a global queue and scheduled to execute on a multiprocessor platform with $m > 1$ identical processors. Tasks are independent, preemptable, and may migrate from one processor to another during execution. Although RUN is intended to minimize preemptions and migrations, our calculations make the standard (incorrect) assumption that these events take zero time. In an actual system, measured preemption and migration overheads can be accommodated by adjusting task worst-case execution times.

Our schedules specify which jobs are running at any given time, without concern for the specific job-to-processor assignment. In an executing schedule, $e(J, t)$ denotes the work remaining for job $J$ at time $t$, so that $e(J, t)$ equals $J.c$ minus the amount of time that $J$ has already executed up until time $t$.

**Definition 4** (Schedule) For a set of jobs (or tasks) $\mathcal{J}$ on a platform of $m$ identical processors, a *schedule* $\Sigma$ is a function from the set of all non-negative times $t$ onto the power set of $\mathcal{J}$ such that (i) $|\Sigma(t)| \leq m$ for all $t$, and (ii) if $J \in \Sigma(t)$, then $J.r \leq t$, and $e(J, t) > 0$. Thus $\Sigma(t)$ represents the set of jobs executing at time $t$.

Condition (ii) requires that we only execute jobs which have been released and have work remaining. Once we have demonstrated that our algorithm produces valid schedules, we will consider the problem of processor assignment (see Sect. 6).

Note also that $\Sigma(t)$ is a *set* of jobs, meaning that no job is selected multiple times in a single time instant. In other words, no job can execute simultaneously on multiple processors. This assignment restriction lies at the heart of the multiprocessor scheduling problem, as observed by Liu (1969) (and quoted in Baruah 2001):

> The simple fact that *a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors.*

**Definition 5** (Valid schedule) A schedule $\Sigma$ of a job set $\mathcal{J}$ is *valid* if all jobs in $\mathcal{J}$ finish by their deadlines.

A set $\mathcal{T}$ of jobs or tasks is *feasible* if a valid schedule for it exists. $\mathcal{T}$ is *schedulable* by an algorithm if the schedule produced for $\mathcal{T}$ by this algorithm is valid. A scheduling algorithm is *optimal* if it finds a valid schedule for all feasible sets of tasks.

### 2.3 Fully utilized system

A system of $m$ processors is *fully utilized* by a task set $\mathcal{T}$ provided that (i) the accumulated rate $\rho(\mathcal{T})$ of $\mathcal{T}$ is equal to $m$; (ii) all jobs always require their full WCET times; and (iii) all tasks release a job a time zero. Henceforth, we will only consider fully utilized systems. We will observe, however, that this assumption does not actually restrict our task model.

First, if the accumulated rate of $\mathcal{T}$ is less than $m$, idle (dummy) tasks may be inserted as needed to make up the difference. When a dummy task is scheduled to "execute", we simply idle its assigned processor. Since a fixed set of dummy tasks is created prior to on-line scheduling, these dummy tasks may be treated exactly like real tasks, both in theory and in simulation. However, if we are careful in the creation and placement of these dummy tasks, we may significantly improve system performance by partially or fully partitioning our task set onto fixed processors (see Sect. 6.1).

Second, assume that a job $J$ has a WCET estimate of $J.c$, but that it completes after consuming only $c' < J.c$ units of processor time. In such a case, the system can easily simulate $J.c - c'$ of $J$'s execution by blocking a processor accordingly. We may thus assume that a job's WCET estimate is always correct, and that each job $J$ executes for exactly $\rho(\tau)(J.d - J.r)$ time during the interval $[J.r, J.d)$.

Third, suppose that some task $\tau$ has its initial job release at some time $s > 0$, and that $s$ is known at the outset. We may then add a dummy job $J_0$ with release time 0, deadline $s$, and execution time $J_0.c = \rho(\tau)s$.

So without loss of generality, we henceforth assume that all systems are fully utilized. One consequence of this is that, in any valid schedule $\Sigma$ on $m$ identical processors, we must have $|\Sigma(t)| = m$ for all times $t$.

We now begin to describe the construction of our optimal RUN scheduling algorithm.

## 3 Servers

RUN's reduction from multiprocessor to uniprocessor systems is enabled by aggregating tasks into *servers*. We treat servers as tasks with a sequence of jobs, but they are not actual tasks in the system; each server is a proxy for a collection of *client* tasks. In any instant when a server is running, its allocated processor time is actually being used by one of its clients. A server's clients are scheduled via some internal scheduling mechanism of the server.

Since we treat servers as tasks, the rate of a server can never be greater than one; consequently, this section focuses only on uniprocessor systems. We precisely define the concept of servers (Sect. 3.1) and show how they correctly schedule the client tasks associated with them (Sect. 3.2). We return to multiprocessors in the following section.

### 3.1 Server model and notations

A server for a set of tasks is defined as follows:

**Definition 6** (Server/client) Let $\mathcal{T}$ be a set of tasks with a total rate given by $\rho(\mathcal{T}) = \sum_{\tau \in \mathcal{T}} \rho(\tau) \leq 1$. A *server* $S$ for $\mathcal{T}$, denoted $\mathrm{ser}(\mathcal{T})$, is a virtual task with rate $\rho(\mathcal{T})$, release times $\mathrm{R}(S) = \bigcup_{\tau \in \mathcal{T}} \mathrm{R}(\tau)$, and some internal scheduling policy to schedule the tasks in $\mathcal{T}$. $\mathcal{T}$ is the set of $S$'s *clients*, and is denoted $\mathrm{cli}(S)$.

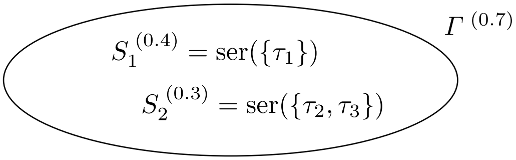

$\Gamma^{(0.7)}$
$S_1^{(0.4)} = \mathrm{ser}(\{\tau_1\})$
$S_2^{(0.3)} = \mathrm{ser}(\{\tau_2, \tau_3\})$

**Fig. 5** A two-server set. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$

Since servers are themselves virtual tasks, we may also speak of a server for a set of servers. And since a server may contain only a single client task, the concepts are largely interchangeable.

We refer to a job of any client of $S$ as a *client job* of $S$. If $S$ is a server and $\Gamma$ a set of servers, then $\mathrm{ser}(\mathrm{cli}(S)) = S$ and $\mathrm{cli}(\mathrm{ser}(\Gamma)) = \Gamma$.

As we will see in Sect. 4.2, the packing of clients into servers is done off-line prior to execution, and remains static during on-line scheduling. It is therefore unambiguous to define the rate $\rho(S)$ of server $S$ to be $\rho(\mathrm{cli}(S))$.

By Definition 6, the execution requirement of a server $S$ in any interval $[r_i, r_{i+1})$ equals $\rho(S)(r_{i+1} - r_i)$, where $r_i$ and $r_{i+1}$ are consecutive release times in $\mathrm{R}(S)$. Then the workload for job $J$ of server $S$ with $J.r = r_i$ and $J.d = r_{i+1}$ equals $J.c = e(J, J.r) = \rho(S)(J.d - J.r)$, just as with a “real” job. However, since a server $S$ is just a proxy for its clients, the jobs of $S$ are just budgets of processor time allocated to $S$ so that its clients may execute. These *budget jobs* of $S$ may be viewed as $S$ simply replenishing its budget for each interval $[r_i, r_{i+1})$. Similarly, if $J$ is the current job of $S$ at time $t$, then $e(J, t)$ is just the budget of $S$ remaining at time $t$.

As an example, consider Fig. 5, where $\Gamma$ is a set comprised of the two servers $S_1 = \mathrm{ser}(\{\tau_1\})$ and $S_2 = \mathrm{ser}(\{\tau_2, \tau_3\})$ for the tasks $\tau_1$, and $\tau_2$ and $\tau_3$, respectively. If $\rho(\tau_1) = 0.4$, $\rho(\tau_2) = 0.2$ and $\rho(\tau_3) = 0.1$, then $\rho(S_1) = 0.4$ and $\rho(S_2) = 0.3$. Also, if $S = \mathrm{ser}(\Gamma)$ is the server in charge of scheduling $S_1$ and $S_2$, then $\Gamma = \mathrm{cli}(S) = \{S_1, S_2\}$ and $\rho(S) = 0.7$.

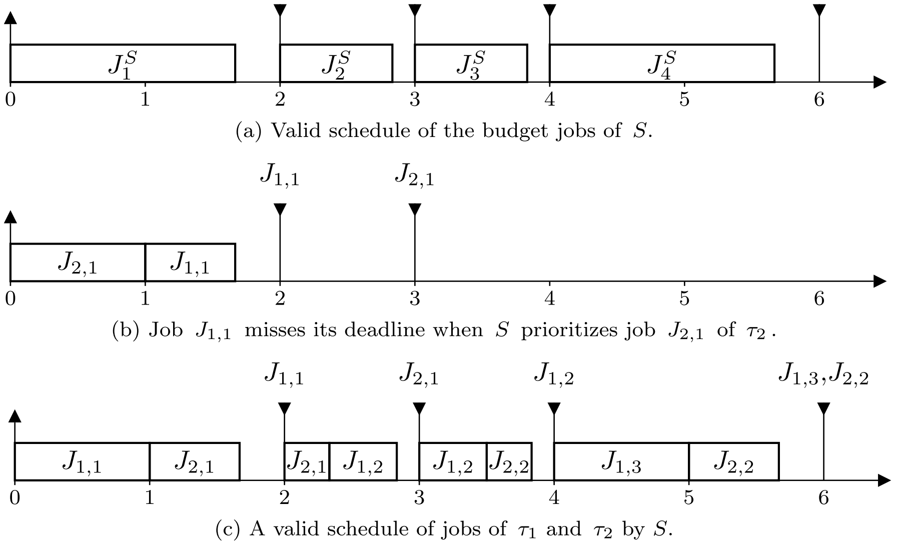
Tasks sets and servers with accumulated rates of one are a key part of RUN’s construction. We therefore define *unit sets* and *unit servers*, both of which can be feasibly scheduled on one processor.

**Definition 7** (Unit set/unit server) A set $\Gamma$ of servers is a *unit set* if $\rho(\Gamma) = 1$. The server $\mathrm{ser}(\Gamma)$ for a unit set $\Gamma$ is a *unit server*.

We say that a server meets its deadlines if all of its budget jobs meet theirs. Even if this is the case, the server must employ an appropriate scheduling policy to ensure that its clients also meet their deadlines.

For example, consider two periodic tasks $\tau_1 : (1/2, 2\mathbb{N})$ and $\tau_2 : (1/3, 3\mathbb{N})$ (rates are $1/2$ and $1/3$, and periods are 2 and 3, respectively, and initial release times are zero). Consider a server $S$ scheduling these two tasks on a dedicated processor. We have $\mathrm{R}(S) = \{0, 2, 3, 4, 6, \ldots\}$ and $\rho(S) = 5/6$. $S$’s first job $J_1^S$ will be released at $J_1^S.r = 0$, and will have deadline $J_1^S.d = 2$ and workload $e(J_1^S, 0) = \rho(S)(2 - 0) = 5/3$, i.e., $S$ has a budget of $5/3$ for the interval $[0, 2)$.

Figure 6(a) shows a valid schedule for the first four budget jobs of $S$. But suppose that $S$ employs a scheduling policy for its clients where $\tau_2$ is given priority over $\tau_1$ (see Fig. 6(b), where $J_{i,j}$ represents the $j$th job of $\tau_i$). Then $J_{2,1}$ will consume one



(a) Valid schedule of the budget jobs of $S$.

(b) Job $J_{1,1}$ misses its deadline when $S$ prioritizes job $J_{2,1}$ of $\tau_2$ .

(c) A valid schedule of jobs of $\tau_1$ and $\tau_2$ by $S$.

**Fig. 6** Schedule of $\tau_1$:(1/2, 2ℕ) and $\tau_2$:(1/3, 3ℕ) by a single server $S$ on a dedicated processor, where R($S$) = {2, 3, 4, 6, ...} and $\rho(S) = 5/6$

unit of time before $J_{1,1}$ begins its execution. The remaining server budget $e(J_1^S, 1) = 2/3$ will be insufficient to complete $J_{1,1}$'s workload of 1 by its deadline at time 2. We see that a server meeting its deadlines does not insure that its clients will meet theirs.

If, on the other hand, $S$'s scheduling policy had prioritized $\tau_1$ at time zero, this deadline miss would be avoided. This is the case with the optimal EDF scheduling policy, which is shown in Fig. 6(c).

## 3.2 EDF server

We will henceforth use EDF as our servers' scheduling policy, as it is optimal, simple, and efficient (Baruah and Goossens 2004; Liu and Layland 1973). A server which schedules its client jobs via EDF is referred to as an *EDF server*.

**Rule 1** (EDF servers) *A server will simulate a uniprocessor system, and will schedule its client jobs using EDF.*

As an illustrative example, consider a set of two periodic tasks $\mathcal{T} = \{\tau_1 : (1/2, 2\mathbb{N}), \tau_2 : (1/3, 3\mathbb{N})\}$. Since $\rho(\mathcal{T}) = 5/6 \leq 1$, we can define an EDF server $S$ to schedule $\mathcal{T}$ such that $\mathrm{cli}(S) = \mathcal{T}$ and $\rho(S) = 5/6$. Figure 7 shows both the evolution of $e(J_S, t)$ during interval $[0, 6)$ and the schedule $\Sigma$ of $\mathcal{T}$ by $S$ on a single processor. In this figure, $J_{i,j}$ represents the $j$th job of $\tau_i$. During intervals $[5/3, 2)$, $[2 + 5/6, 3 + 1/6)$, $[4 + 2/3, 4 + 5/6)$ and $[5 + 1/3, 5 + 1/2)$ (shown with crosshatching), $S$ does not execute, either because its budget is exhausted, or some external job of higher priority has preempted it.

Note that a unit EDF server $S$ has rate $\rho(S) = 1$ and must execute continuously in order to meet its clients' deadlines. Deadlines of $S$ have no effect, since budgets are

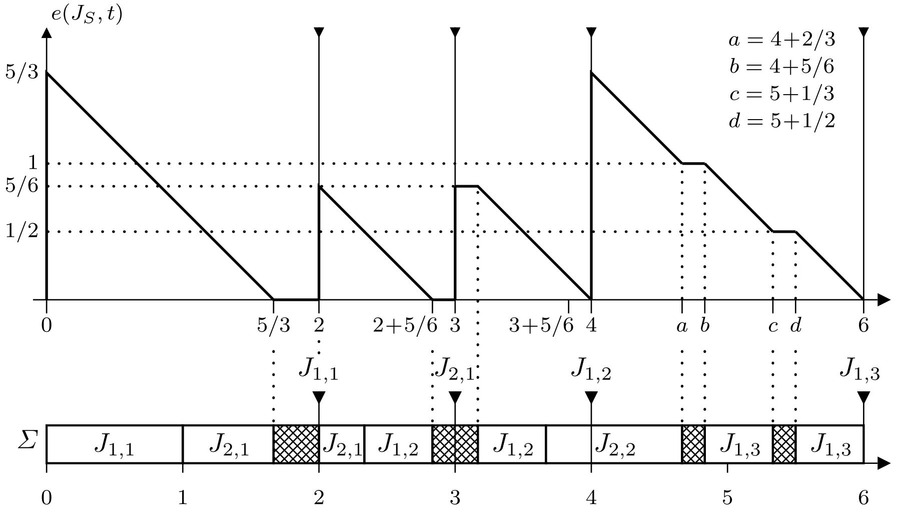


**Fig. 7** Budget management and schedule of $S$; $\mathrm{cli}(S) = \{\tau_1 :(1/2, 2\mathbb{N}), \tau_2 :(1/3, 3\mathbb{N})\}$ and $\rho(S) = 5/6$; crosshatched regions represent intervals when $S$ does not execute, due either to an exhausted budget or preemption by higher priority external jobs

replenished (a new budget job arrives) the instant they are depleted (at the old budget job's deadline).

**Theorem 1** *The EDF server $S = \mathrm{ser}(\Gamma)$ of a set of servers $\Gamma$ produces a valid schedule of $\Gamma$ when $\rho(\Gamma) \leq 1$ and all jobs of $S$ meet their deadlines.*

*Proof* By treating the servers in $\Gamma$ as tasks, we can apply well known results for scheduling task systems. For convenience, we assume that $S$ executes on a single processor; this need not be the case in general, as long as $S$ does not execute on multiple processors in parallel.

Recall from Definition 6 that $\rho(\Gamma) = \sum_{S_i \in \Gamma} \rho(S_i)$. We first prove the theorem for $\rho(\Gamma) = 1$,then use this result for the case of $\rho(\Gamma) < 1$.

*Case* $\rho(\gamma) = 1$: Let $\eta_\Gamma(t, t')$ be the execution demand within a time interval $[t, t')$, where $t < t'$. This demand gives the sum of all execution requests (i.e., jobs) that are released no earlier than $t$ and with deadlines no later than $t'$. By Definition 6, this quantity is bounded above by

$$\eta_\Gamma(t, t') \leq (t' - t) \sum_{S_i \in \Gamma} \rho(S_i)$$

and because $\sum_{S_i \in \Gamma} \rho(S_i) = 1$ by assumption,

$$\eta_\Gamma(t, t') \leq (t' - t). \tag{1}$$

It is known that there is no valid schedule for $\Gamma$ if and only if there is some interval $[t, t')$ such that $\eta_\Gamma(t, t') > t' - t$ (Baruah and Goossens 2004; Baruah et al. 1990).

Since (1) implies that this cannot happen, some valid schedule for $\Gamma$ must exist. Because $S$ schedules $\Gamma$ using EDF and EDF is optimal, $S$ must produce a valid schedule.

*Case* $\rho(\gamma) < 1$: In order to use the result for case $\rho(\Gamma) = 1$, we introduce a slack-filling task $\tau'$, as illustrated in Fig. 7, where $\mathrm{R}(\tau') = \mathrm{R}(S)$ and $\rho(\tau') = 1 - \rho(S)$. We let $\Gamma' = \Gamma \cup \{\tau'\}$, and let $S'$ be an EDF server for $\Gamma'$. Since $\rho(\Gamma') = 1$, $S'$ produces a valid schedule for $\Gamma'$.

Now consider the scheduling window $W_J = [J.r, J.d]$ for some budget job $J$ of $S$. Since $\mathrm{R}(\tau') = \mathrm{R}(S)$, $\tau'$ also has a job $J'$ where $J'.r = J.r$ and $J'.d = J.d$. Since $S'$ produces a valid schedule, $\tau'$ and $S$ do exactly $\rho(\tau')(J.d - J.r)$ and $\rho(S)(J.d - J.r)$ units of work, respectively, during $W_J$. Since there are no deadlines or releases between $J.r$ and $J.d$, the workload of $\tau'$ may be arbitrarily rearranged or subdivided within the interval $W_J$ without compromising the validity of the schedule. We may do this within all scheduling windows of $S$ so as to reproduce *any* schedule of $S$ where it meets its deadlines. Finally, since $S$ and $S'$ both schedule tasks in $\Gamma$ with EDF, $S$ will produce the same *valid* schedule for $\Gamma$ as $S'$, giving our desired result. □

As noted above, a server and its clients may migrate between processors, as long as no more than one client executes at a time. This will allow us to schedule multiple servers on a multiprocessor platform.

### 3.3 Deadline sharing

Unlike previous proportionate fairness approaches to optimal scheduling, client tasks scheduled by servers do not receive their proportional shares of work between each system deadline, nor even between each server deadline. Instead, because a server shares all the deadlines of its clients, we need only ensure that each server receives its "fair" allocation of bandwidth between its deadlines; the server is responsible for correctly distributing that bandwidth among its clients. This *Deadline Sharing* approach requires that each server is:

1. Guaranteed a budget proportional to the sum of its clients' rates between any two consecutive deadlines of its clients;
2. Responsible for scheduling its clients in some correct fashion (e.g., EDF) between such deadlines.

Thus, according to Theorem 1, Deadline Sharing scheduling guarantees the correct scheduling of a server's clients. This approach applies much weaker over-constraints to the system than traditional proportionate fairness, and thus requires significantly fewer preemptions and migrations for optimal scheduling.

### 3.4 Partial knowledge

Unlike periodic tasks, fixed-rate tasks do not, and need not, make all of their release times known at the outset. It is sufficient that they have implicit deadlines, i.e., that there are neither gaps nor overlaps between a task's consecutive jobs. In order for a server to set the deadline for its next budget job, it only needs to know the next deadline of each of its clients. This is also sufficient for a EDF server to make its scheduling selections from among its clients. Thus, unlike periodic tasks, which implicitly provide *all* their release times at the outset, fixed-rate tasks need only make their *next* release time known in order to be scheduled on-line by an EDF server.

## 4 RUN off-line reduction

In this section, we describe the operations DUAL and PACK, which are used iteratively in an off-line procedure to reduce a multiprocessor task system into a collection of uniprocessor systems. In the following section we will show how the EDF schedules for these uniprocessor systems may be transformed back into a schedule for the original multiprocessor system.

The DUAL operation, detailed in Sect. 4.1, transforms a server $S$ into the dual server $S^*$, whose execution time represents the idle time of $S$. Since $\rho(S^*) = 1 - \rho(S)$, the DUAL operation reduces the total rate and the number of required processors in systems where most servers have high rates.

Such high rate servers are generated via the PACK operation, presented in Sect. 4.2. A set of servers whose rates sum to no more than one can be packed into a new, aggregated server, reducing the total number of servers and producing a high-rate server favored by the DUAL operation. Given this synergy, we compose the two operations into a single REDUCE operation, which will be defined in Sect. 4.3. As will be seen in Sect. 5, after an off-line sequence of REDUCE operations, the on-line schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems.

### 4.1 DUAL operation

The simple duality example in Sect. 1.2 demonstrates a special case wherein $m + 1$ tasks are to be scheduled on $m$ processors. In such a case, the dual task set has an accumulated rate of one, and may therefore be scheduled on a single processor, as originally established in Levin et al. (2009). The primal schedule is then easily inferred, as shown in Fig. 2. Dual Scheduling Equivalence (DSE) uses aggregated servers to extend these results to more general task systems.

**Definition 8** (Dual server) The *dual server* $S^*$ of a server $S$ is a server with the same deadlines as $S$ and with rate $\rho(S^*)$ equal to $1 - \rho(S)$. If $\Gamma$ is a set of servers, then its dual set $\Gamma^*$ is the set of dual servers to those in $\Gamma$, i.e., $S \in \Gamma$ if and only if $S^* \in \Gamma^*$.

A unit server, which has rate $\rho(S) = 1$ and must execute continuously in order to meet its clients' deadlines, has as its dual a *null server*, which has rate $\rho(S^*) = 0$ and never executes.

**Definition 9** (Dual schedule) Let $\Gamma$ be a set of servers and $\Gamma^*$ be its dual set. Two schedules $\Sigma$ of $\Gamma$ and $\Sigma^*$ of $\Gamma^*$ are duals if, for all times $t$ and all $S \in \Gamma$, $S \in \Sigma(t)$ if and only if $S^* \notin \Sigma^*(t)$; that is, $S$ executes exactly when $S^*$ is idle, and vice versa.

The $S$, $\Gamma$, and $\Sigma$ are referred to as *primal* relative to their duals $S^*$, $\Gamma^*$, and $\Sigma^*$. As with any good notion of "duality", we find that $(S^*)^* = S$, $(\Gamma^*)^* = \Gamma$ and $(\Sigma^*)^* = \Sigma$. In fact, it is this property of dual schedules that motivated the avoidance of task-to-processor assignment in Definition 4. Here we are only concerned with *which* tasks (or servers) are executing at any given time; task-to-processor assignment may be handled as a separate step (see Sect. 6.1).

We now establish Dual Scheduling Equivalence (DSE) which states that the schedule of a primal set of servers is correct precisely when its dual schedule is correct.

**Theorem 2** (Dual scheduling equivalence) *Let $\Gamma$ be a set of $n = m + k$ servers with $k \geq 1$ and $\rho(\Gamma) = m$, an integer. For a schedule $\Sigma$ of $\Gamma$ on $m$ processors, let $\Sigma^*$ and $\Gamma^*$ be their respective duals. Then $\rho(\Gamma^*) = k$, and so $\Gamma^*$ is feasible on $k$ processors. Further, $\Sigma$ is valid if and only if $\Sigma^*$ is valid.*

*Proof* First,

$$\rho(\Gamma^*) = \sum_{S^* \in \Gamma^*} \rho(S^*) = \sum_{S \in \Gamma} (1 - \rho(S)) = n - \rho(\Gamma) = k$$

so $k$ processors are sufficient to schedule $\Gamma^*$. Next, we prove that if $\Sigma$ is valid for $\Gamma$ then Definitions 4 and 5 imply that $\Sigma^*$ is valid for $\Gamma^*$.

Because $\Sigma$ is a valid schedule on $m$ processors and we assume full utilization, $\Sigma$ always executes $m$ distinct tasks. The remaining $k = n - m$ tasks are idle in $\Sigma$, and so are exactly the tasks executing in $\Sigma^*$. Hence by Definition 9, $\Sigma^*$ is always executing exactly $k$ distinct tasks on its $k$ (virtual dual) processors. Since $\Sigma$ is valid, any job $J$ of server $S \in \Gamma$ does exactly $J.c = \rho(S)(J.d - J.r)$ units of work between its release $J.r$ and its deadline $J.d$. During this same time, $S^*$ has a matching job $J^*$ where $J^*.r = J.r$, $J^*.d = J.d$, and

$$\begin{aligned} J^*.c &= \rho(S^*)(J^*.d - J^*.r) \\ &= (1 - \rho(S))(J.d - J.r) \\ &= (J.d - J.r) - J.c. \end{aligned}$$

That is, $J^*$'s execution time during the interval $[J.d, J.r)$ is exactly the length of time that $J$ must be idle. Thus, as $J$ executes for $J.c$ during this interval in $\Sigma$, $J^*$ executes for $J^*.c$ in $\Sigma^*$. Consequently, $J^*$ satisfies condition (ii) of Definition 4 and also meets its deadline. Since this holds for all jobs of all dual servers, $\Sigma^*$ is a valid schedule for $\Gamma^*$.

The converse also follows from the above argument, since $(\Sigma^*)^* = \Sigma$. □

Once again, see Fig. 2 for a simple illustration. The dual tasks/servers, and the virtual processors on which they are scheduled, are collectively known as the *dual system*.

We now summarize this dual scheduling rule for future reference.

**Rule 2** (Dual server) *At any time, execute in $\Sigma$ the servers of $\Gamma$ whose dual servers are not executing in $\Sigma^*$, and vice versa.*

Finally, we define the DUAL operation $\varphi$ from a set of servers $\Gamma$ to its dual set $\Gamma^*$ as the bijection which associates a server $S$ with its dual server $S^*$, i.e., $\varphi(S) = S^*$. We adopt the usual notational convention for the image of a subset. That is, if $f : A \to B$ is a function from $A$ to $B$ and $A' \subseteq A$, we understand $f(A')$ to mean $\{f(a), a \in A'\}$. For example, $\varphi(\Gamma) = \{S^*, S \in \Gamma\} = \Gamma^*$.

Note that Theorem 2 does not provide any particular rules for generating a schedule; it merely establishes the equivalence of scheduling $n$ tasks on $m$ processors with scheduling their dual tasks on $n - m$ virtual processors. This can be advantageous when $n - m < m$ and the number of processors is reduced, as seen in Fig. 2. The PACK operation ensures this desirable outcome.

### 4.2 PACK operation

We cannot generally expect to find $n - m < m$. Consider the example of a set $\mathcal{T}$ of 5 tasks, each with rate 2/5. Here, $n = 5$, $m = \rho(\mathcal{T}) = 2$, and $n - m = 3 > 2$. The dual tasks in $\mathcal{T}^*$ have rates of 3/5, and will require 3 processors to schedule. Here, the DUAL operation has made the system larger, not smaller.

However, suppose we combine two pairs of tasks in $\mathcal{T}$ into two new tasks (servers), each with rates of $2/5 + 2/5 = 4/5$. Then the dual of this new 3 task set has rates $1/5$, $1/5$, and $3/5$, and may be scheduled on a single processor. This is because this dual system requires $n - m$ processors, and we have just reduced $n$ by two without changing $m$. This is the essence of the PACK operation.

**Definition 10** (Packing) Let $\Gamma$ be a set of servers. A partition $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ of $\Gamma$ is a *packing* of $\Gamma$ if $\rho(\Gamma_i) \leq 1$ for all $i$ and $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ for all $i \neq j$. An algorithm $A$ is a *packing algorithm* if it partitions any set of servers into a packing. In such a case, we denote the packing of $\Gamma$ produced by $A$ as $\pi_A[\Gamma]$.

An example of packing a set $\Gamma$ of 7 servers into three sets $\Gamma_1$, $\Gamma_2$ and $\Gamma_3$, is illustrated by rows 1 and 2 of Fig. 8.

Standard bin-packing algorithms suffice to create packings of server sets, where the items being packed are servers (with sizes equal to rates), and bins (new aggregate servers) have a capacity of 1. They proceed by "packing" items one at a time, either in random or decreasing order. If the item fits in some existing bin, it is placed there; otherwise, it is placed in a new, empty bin. Differences between algorithms come from the tie-breaking scheme when an item will fit in multiple existing bins. *First-fit* examines bins in the order they were created, and puts the item into the first bin in which it fits; *worst-fit* places the item into the bin with the least remaining capacity into which it still fits; *best-fit* places the item into the bin with the most remaining capacity, if it fits. Any of these schemes will create a packing of any set of servers.

**Theorem 3** *The first-fit, worst-fit and best-fit bin-packing algorithms are packing algorithms.*

*Proof* At any step of these three algorithms, a new bin can only be created if the current task to be allocated does not fit in any of the existing partially filled bins. Now

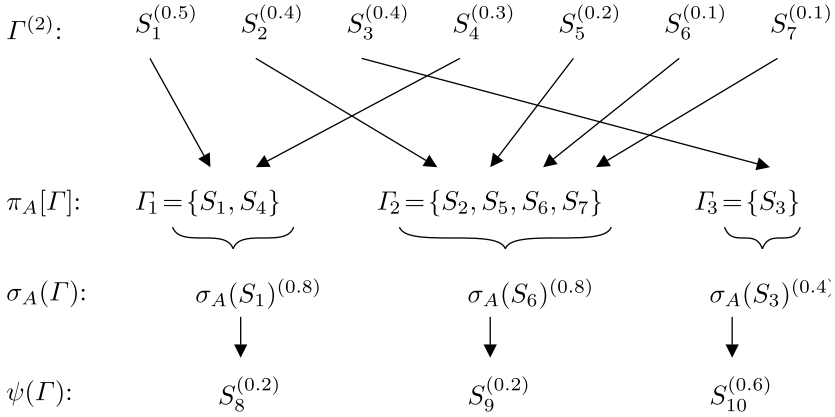


**Fig. 8** Packing, PACK and DUAL operations applied to $\Gamma = \{S_1, S_2, \ldots, S_7\}$, resulting in a reduction to a unit set of three servers $\{S_8, S_9, S_{10}\}$ with $S_8 = \varphi \circ \sigma_A(S_1)$, $S_9 = \varphi \circ \sigma_A(S_6)$, $S_{10} = \varphi \circ \sigma_A(S_3)$. The notation $X^{(\mu)}$ means that $\rho(X) = \mu$

suppose that $\rho(\Gamma_i) + \rho(\Gamma_j) \leq 1$ for some two bins, where $\Gamma_j$ was created after $\Gamma_i$. Then the first item $\tau$ placed in $\Gamma_j$ must have $\rho(\tau) \leq \rho(\Gamma_j) \leq 1 - \rho(\Gamma_i)$. That is, $\tau$ fits in bin $\Gamma_i$, contradicting the need to create $\Gamma_j$ for it. Therefore $\rho(\Gamma_i) + \rho(\Gamma_j) > 1$ must hold for any pair of bins. □

Hereafter, we assume that $A$ is a packing algorithm. Since $\pi_A[\Gamma]$ is a partition of $\Gamma$, it induces an equivalence relation between servers in $\Gamma$ whose equivalence classes are the elements in $\pi_A[\Gamma]$.

We wish to assign an aggregated server to schedule each equivalence class in $\pi_A[\Gamma]$. To this end, we first introduce $p_A$, the canonical mapping of $\Gamma$ onto $\pi_A[\Gamma]$, which maps a server in $\Gamma$ to its equivalence class in $\pi_A[\Gamma]$, i.e., $p_A(S) = p_A(S')$ if and only if $S, S' \in \Gamma_i$ for some $\Gamma_i \in \pi_A[\Gamma]$. Then, we define the PACK operation.

**Definition 11** (PACK operation) Let $\Gamma$ be a set of servers, $A$ a packing algorithm, and $\pi_A[\Gamma]$ the resultant packing. For each $\Gamma_i \in \pi_A[\Gamma]$, we assign it a dedicated server $\mathrm{ser}(\Gamma_i)$. The PACK *operation* $\sigma_A$ is the mapping from $\Gamma$ onto $\mathrm{ser}(\pi_A[\Gamma])$ defined by $\sigma_A = \mathrm{ser} \circ p_A$, where $p_A$ is the canonical mapping from $\Gamma$ onto $\pi_A[\Gamma]$ and $\mathrm{ser}(\pi_A[\Gamma]) = \{\mathrm{ser}(\Gamma_i), \Gamma_i \in \pi_A[\Gamma]\}$. Hence, $\sigma_A$ associates a server $S \in \Gamma$ with the server $\sigma_A(S)$ in $\mathrm{ser}(\pi_A[\Gamma])$ responsible for scheduling $p_A(S)$.

According to this definition, if $S$ and $S'$ are packed in the same subset $\Gamma_i$ by packing algorithm $A$, then $\sigma_A(S) = \sigma_A(S')$. By our notational conventions, $\sigma_A(\Gamma) = \{\mathrm{ser}(\Gamma_i), \Gamma_i \in \pi_A[\Gamma]\}$ is the set of aggregated servers responsible for scheduling each of the equivalence classes of $\pi_A[\Gamma]$.

For example, Fig. 8 shows that $\sigma_A(S_6) = \mathrm{ser}(\Gamma_2)$ is the aggregated server responsible for scheduling all the servers in $\Gamma_2$, and that $\sigma_A(\Gamma) = \{\sigma_A(S_1), \sigma_A(S_6), \sigma_A(S_3)\}$.

**Definition 12** (Packed server set) A set of servers $\Gamma$ is *packed* if it is a singleton, or if $|\Gamma| \geq 2$ and for any two distinct servers $S$ and $S'$ in $\Gamma$, $\rho(S) + \rho(S') > 1$ and $\mathrm{cli}(S) \cap \mathrm{cli}(S') = \{\}$.

Consequently, the packing of a packed server set $\Gamma$ is the collection of singleton sets $\{\{S\}\}_{S \in \Gamma}$.

In this paper, we are only concerned with packings that result from the application of some packing algorithm, so we will henceforth drop from the implicitly understood $_A$ from $\pi[\Gamma]$ and $\sigma$.

### 4.3 REDUCE operation

We now compose the DUAL and PACK operations into the REDUCE operation. As will be shown, a sequence of reductions transforms a multiprocessor scheduling problem into a collection of uniprocessor scheduling problems. This off-line transformation is the cornerstone of the RUN algorithm.

**Lemma 1** *Let $\Gamma$ be a packed set of servers, and let $\varphi(\Gamma)$ be the dual set of $\Gamma$. Suppose we apply a* PACK *operation $\sigma$ to $\varphi(\Gamma)$. Then*

$$\left|\sigma \circ \varphi(\Gamma)\right| \leq \left\lceil \frac{|\Gamma|}{2} \right\rceil.$$

*Proof* Since $\Gamma$ is packed, then for any $S_i, S_j \in \Gamma$, we have $\rho(S_i) + \rho(S_j) > 1$. If $S_i^*, S_j^* \in \varphi(\Gamma)$ are their duals, then

$$\rho\big(S_i^*\big) + \rho\big(S_j^*\big) = \big(1 - \rho(S_i)\big) + \big(1 - \rho(S_j)\big) = 2 - \big(\rho(S_i) + \rho(S_j)\big) < 2 - 1 = 1.$$

That is, any two dual servers will fit into a bin together. So any packing $\sigma$ of $\varphi(\Gamma)$ will, at a minimum, pair off the dual servers (with one leftover if $|\varphi(\Gamma)|$ is odd). Hence, $|\sigma \circ \varphi(\Gamma)| \leq \lceil |\Gamma|/2 \rceil$. □

Thus, packing the dual of a packed set reduces the number of servers by at least (almost) half. Since we will use this pair of operations repeatedly, we define a REDUCE operation to be their composition.

**Definition 13** (REDUCE operation) Given a set of servers $\Gamma$ and a packing algorithm $A$, a REDUCE *operation* on a server $S$ in $\Gamma$, denoted $\psi(S)$, is the composition of the DUAL operation $\varphi$ with the PACK operation $\sigma$ associated with $A$, i.e., $\psi(S) = \varphi \circ \sigma(S)$.

Figure 8 illustrates the steps of the REDUCE operation $\psi$. As we intend to apply REDUCE repeatedly until we are left with only unit servers, we now define a *reduction sequence*.

**Definition 14** (Reduction level/sequence) Let $i \geq 1$ be an integer, $\Gamma$ a set of servers, and $S$ a server in $\Gamma$. The operator $\psi^i$ is recursively defined by $\psi^0(S) = S$ and

**Table 2** Sample reduction and proper subsets

| | Server Rate | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\psi^0(\Gamma)$ | 0.6 | 0.6 | 0.6 | 0.6 | 0.6 | 0.8 | 0.6 | 0.6 | 0.5 | 0.5 |
| $\sigma(\psi^0(\Gamma))$ | 0.6 | 0.6 | 0.6 | 0.6 | 0.6 | 0.8 | 0.6 | 0.6 | **1**→ | |
| $\psi^1(\Gamma)$ | 0.4 | 0.4 | 0.4 | 0.4 | 0.4 | 0.2 | 0.4 | 0.4 | 0 | |
| $\sigma(\psi^1(\Gamma))$ | 0.8 | | 0.8 | | 0.4 | **1**→ | | | | |
| $\psi^2(\Gamma)$ | 0.2 | | 0.2 | | 0.6 | 0 | | | | |
| $\sigma(\psi^2(\Gamma))$ | **1** | | | | | | | | | |

$\psi^i(S) = \psi \circ \psi^{i-1}(S)$. $\{\psi^i\}_i$ is a *reduction sequence*, and the server system $\psi^i(\Gamma)$ is said to be at *reduction level* $i$.

Theorem 4 will show that a reduction sequence on a server set $\Gamma$ with $\rho(\Gamma) = m$ always arrives at a collection of unit servers. As we shall see, a unit server may or may not have its dual rolled into further reductions. Any unit server which is *not* further reduced (i.e., which terminates the reduction sequence of some subset of tasks) is called a *terminal unit server*. Table 2 shows 10 tasks (or servers) transformed into a unit server via two REDUCE operations and a final PACK. Notice that two unit servers appear before the final reduction level (indicated in the table by **1**→). A subset of tasks in $\Gamma$ which eventually gets reduced to a terminal unit server is referred to as a *proper subset*. A proper subset always has an integral accumulated rate, and may be scheduled on a subset of processors independently from the rest of $\Gamma$. A proper subset, all the intermediate primal and dual servers down to and including the terminal unit server, and the processor(s) assigned to them, are collectively known as a *proper subsystem*. Table 2 may be considered to have one or three proper subsystems, depending on how we choose to treat the intermediate unit servers.

When intermediate unit servers are encountered prior to the final reduction level, there are two ways of dealing with them: we may isolate them or ignore them. Under the first approach, when an intermediate unit server is found, it is treated as a terminal unit server. The proper subsystem that generated it is isolated from the rest of the system, assigned its own processors, and scheduled independently. For example, in Table 2, the proper subset $\{0.8, 0.6, 0.6\}$ would be assigned two processors, which would be scheduled independently from the other four processors and seven tasks. This approach is more efficient in practice, because these independent subsystems do not impose events on, or migrate into, the rest of the system. The simulations detailed in Sect. 6.6 use this approach.

We may also deal with intermediate unit servers by ignoring them. The dual of a unit server is a null server. If we ignore the fact that we have found an intermediate unit server, then its dual null server simply gets packed into some other server in the next level. In Table 2, observe the unit server that results from the tasks $\{0.5, 0.5\}$. The unit server's dual has rate 0, and may be packed along with the dual servers $\{0.2, 0.4, 0.4\}$ into another unit server. Under this approach, we do not consider these intermediate unit servers to be *terminal* unit servers, nor do we consider them to be the root of a proper subsystem. Under this view, Table 2 contains only one terminal unit server and one proper subsystem. It is still possible for a system to have more than one terminal unit server, but only if they all appear at the same final reduction level. For the remainder of this section and the next, we will adopt this approach for dealing with intermediate unit servers, because it simplifies our exposition and proofs.

We now provide two intermediate results which will be used to establish Theorem 4.

**Lemma 2** *Let $\Gamma$ be a set of servers, and let $\sigma(\Gamma)$ be the set of servers assigned to the packing $\pi[\Gamma]$ of some* PACK *operation on $\Gamma$. Then $\rho(\Gamma) \leq |\sigma(\Gamma)|$. Further, if not all servers in $\sigma(\Gamma)$ are unit servers, then $\rho(\Gamma) < |\sigma(\Gamma)|$.*

*Proof* Since $\rho(S) \leq 1$ for all servers $S \in \sigma(\Gamma)$,

$$\rho(\Gamma) = \sum_{\Gamma_i \in \pi[\Gamma]} \rho(\Gamma_i) = \sum_{S \in \sigma(\Gamma)} \rho(S) \leq \sum_{S \in \sigma(\Gamma)} 1 = \left|\sigma(\Gamma)\right|.$$

If not all servers in $\sigma(\Gamma)$ are unit servers, then $\rho(S) < 1$ for some $S \in \sigma(\Gamma)$, and the inequality above is strict. □

**Lemma 3** *Let $\Gamma$ be a packed set of servers, not all of which are unit servers. If $\rho(\Gamma)$ is a positive integer, then $|\Gamma| \geq 3$.*

*Proof* If $\Gamma = \{S_1\}$ and $S_1$ is not a unit server, then $\rho(\Gamma) < 1$, not a positive integer. If $\Gamma = \{S_1, S_2\}$ is a packed set, then $\rho(\Gamma) = \rho(S_1) + \rho(S_2) > 1$; but $\rho(\Gamma)$ is not 2 unless $S_1$ and $S_2$ are both unit servers. Thus $|\Gamma|$ is not 1 or 2. □

**Theorem 4** (Reduction convergence) *Let $\Gamma$ be a set of servers where $\rho(\Gamma)$ is a positive integer. Then for some $p \geq 0$, $\sigma(\psi^p(\Gamma))$ is a set of unit servers.*

*Proof* Let $\Gamma^k = \psi^k(\Gamma)$ and $\Gamma_\sigma^k = \sigma(\Gamma^k)$, and suppose that $\rho(\Gamma_\sigma^k)$ is a positive integer. If $\Gamma_\sigma^k$ is a set of unit servers, then $p = k$ and we are done.

Otherwise, according to Lemma 3, $|\Gamma_\sigma^k| \geq 3$. Observe that

$$\begin{aligned} \Gamma_\sigma^{k+1} &= \sigma\left(\Gamma^{k+1}\right) \\ &= \sigma \circ \psi\left(\Gamma^k\right) \\ &= \sigma \circ \varphi \circ \sigma\left(\Gamma^k\right) \\ &= (\sigma \circ \varphi)\left(\Gamma_\sigma^k\right). \end{aligned}$$

Since $\Gamma_\sigma^k$ is a packed set of servers, Lemma 1 tells us that

$$\left|\Gamma_\sigma^{k+1}\right| \leq \left\lceil \frac{|\Gamma_\sigma^k| + 1}{2} \right\rceil.$$

Since $\lceil (x+1)/2 \rceil < x$ whenever $x \geq 3$, and we know $|\Gamma_\sigma^k| \geq 3$, it follows that

$$\left|\Gamma_\sigma^{k+1}\right| < \left|\Gamma_\sigma^k\right|.$$

**Table 3** Reduction example with different outcomes

| | First Packing | | | | |
|---|---|---|---|---|---|
| $\psi^0(\Gamma)$ | .4 | .4 | .2 | .2 | .8 |
| $\sigma(\psi^0(\Gamma))$ | .8 | | .4 | | .8 |
| $\psi^1(\Gamma)$ | .2 | | .6 | | .2 |
| $\sigma(\psi^1(\Gamma))$ | 1 | | | | |

| Second Packing | | | | |
|---|---|---|---|---|
| .4 | .4 | .2 | .8 | .2 |
| 1 | | | 1 | |

Note that packing a set does not change its rate, so $\rho(\Gamma^k) = \rho(\Gamma^k_\sigma)$. We have assumed that $\rho(\Gamma^k_\sigma)$ is a positive integer, and that $\Gamma^k_\sigma$ are not all unit servers, so Lemma 2 tells us that $\rho(\Gamma^k_\sigma) = \rho(\Gamma^k) < |\Gamma^k_\sigma|$. By setting $m = \rho(\Gamma^k_\sigma)$ and $n = |\Gamma^k_\sigma|$, so that $m < n$, we may apply Theorem 2 to the dual of $\Gamma^k_\sigma$ to deduce that $\rho(\varphi(\Gamma^k_\sigma)) = \rho(\Gamma^{k+1}) = \rho(\Gamma^{k+1}_\sigma)$ is also a positive integer.

We now see that $\Gamma^{k+1}_\sigma$ also has positive integer rate, but contains fewer servers than $\Gamma^k_\sigma$. Hence, starting with the packed set $\Gamma^0_\sigma = \sigma(\Gamma)$, each iteration of $\sigma \circ \varphi$ either produces a set of unit servers or a smaller set with positive integer rate. This iteration can only occur a finite number of times, and once $|\Gamma^k_\sigma| < 3$, Lemma 3 tells us that $\Gamma^k_\sigma$ must be a set of unit servers; we have found our $p = k$, and are done. □

In other words, a reduction sequence on any set of servers eventually produces a set of unit servers. We will show how to schedule the proper subsystem of each unit server in the next section. First, note that the behavior of the $\psi$ operator is dependent on the packing algorithm associated with its PACK operation $\sigma_A$. For example, Table 3 shows two packings of the same set of servers. One produces one unit server after one reduction level and the other produces two unit servers with no reductions. While some packings may be "better" than others (i.e., lead to a more efficient schedule), Theorem 4 implicitly proves that all packing algorithms "work"; they all lead to a correct reduction to *some* set of unit servers.

## 5 RUN on-line scheduling

Now that we have transformed a multiprocessor system into one or more uniprocessor systems, we show how the schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems. First we schedule the clients of the terminal unit servers using EDF. We then iteratively filter this schedule backwards through the reduction hierarchy, using Dual Scheduling Equivalence at DUAL levels, and EDF at PACK levels. This comprises the on-line scheduling portion of our optimal RUN algorithm.

Theorem 4 says that a reduction sequence produces a collection of one or more terminal unit servers. As shown in Table 2, the original task set may be partitioned into the proper subsystems associated with these unit servers, which may then be scheduled independently. So without loss of generality, we assume in this section that $\mathcal{T}$ is a proper subset, i.e., that it is handled by a single terminal unit server at the final reduction level.

The scheduling process is illustrated by inverting the reduction tables from the previous section and creating a *server tree* whose nodes are the servers generated by iterations of the PACK and DUAL operations. The terminal unit server becomes the root of the server tree, which represents the top-level virtual uniprocessor system. The root's children are the unit server's clients, which are scheduled by EDF.

As an illustrative example, let us consider the first proper subsystem in Table 2. To these 5 tasks with rates of 3/5, we will assign periods of 5, 10, 15, 10, and 5. Then our initial task set becomes

$$\mathcal{T} = \{S_1{:}[3/5, 5\mathbb{N}],\ S_2{:}[3/5, 10\mathbb{N}],\ S_3{:}[3/5, 15\mathbb{N}],\ S_4{:}[3/5, 10\mathbb{N}],\ S_5{:}[3/5, 5\mathbb{N}]\}.$$

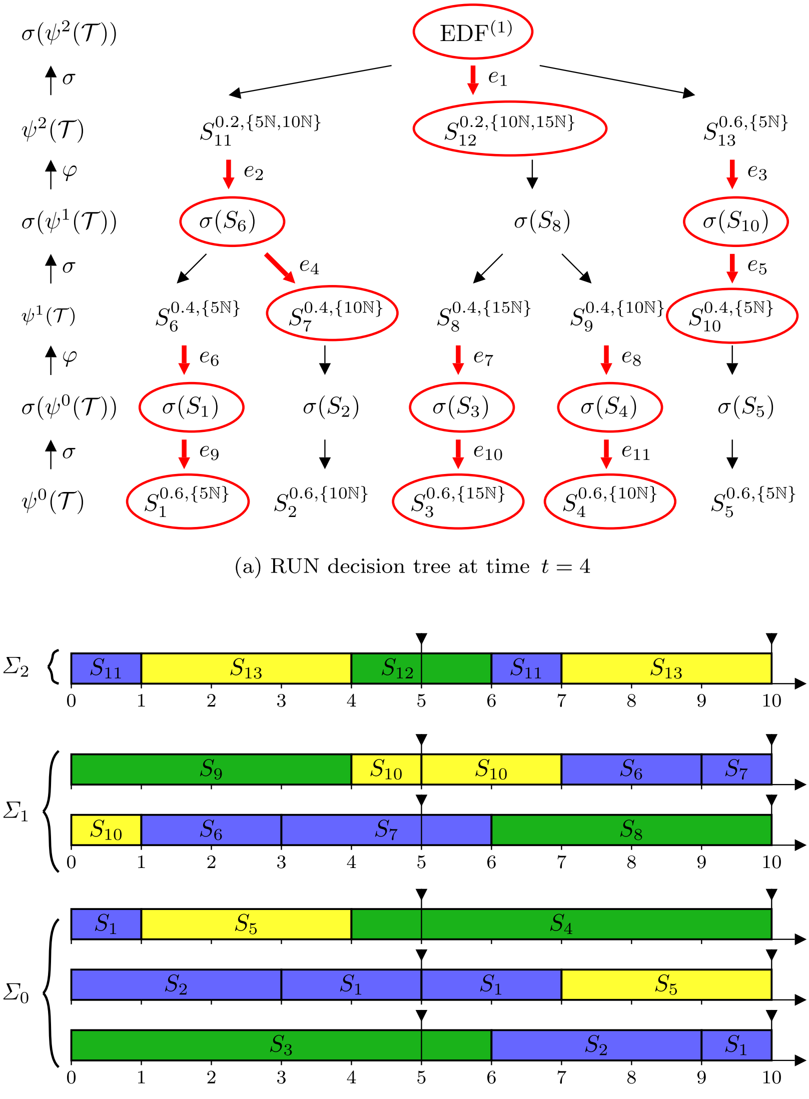
Figure 9(a) shows the inverted server tree, with these 5 tasks as the leaves, and the terminal unit server as the root. We demonstrate the scheduling process that occurs at time $t = 4$ by circling the servers chosen for execution. First, the terminal unit server schedules its children using EDF. At $t = 4$, only the child node $S_{12}$ has any work remaining, so it is chosen to execute (circled). After this, we propagate the circles down the tree using Rules 1 (schedule clients with EDF) and 2 (DSE). For convenience, we restate these here in terms of the server tree:

**Rule 1** (EDF server) *If a packed server is executing* (*circled*), *execute the child node with the earliest deadline among those children with work remaining*; *if a packed server is not executing* (*not circled*), *execute none of its children.*

**Rule 2** (Dual server) *Execute* (*circle*) *the child* (*packed server*) *of a dual server if and only if the dual server is* not *executing* (*not circled*).

Let us continue to follow the branch beneath $S_{12}$. $S_{12}$ is the dual of $\sigma(S_8)$, which will be idle since $S_{12}$ is executing. Since $\sigma(S_8)$ is not executing (not circled), neither are either of its clients. However, these clients, $S_8$ and $S_9$, are the duals of $\sigma(S_3)$ and $\sigma(S_4)$, respectively, so these two servers will be executing (circled). Finally, $\sigma(S_3)$ and $\sigma(S_4)$ are each servers for a single client, so each of these clients, namely $S_3$ and $S_4$, will also execute. The full schedule for all levels of the system is shown through time $t = 10$ in Fig. 9(b), and indeed, we see that $S_3$ and $S_4$ are executing at time $t = 4$. By similarly propagating the circling of nodes down the tree (Fig. 9(a)), we see that $S_1$ also executes at $t = 4$, while $S_2$ and $S_5$ are idle.

In practice, we only need to invoke the above scheduler when some subsystem's EDF scheduler generates a *work complete event* (W.C.E.) *or a job release event* (J.R.E.). In fact, when some EDF server's client has a W.C.E., we only need to propagate the changes down from that server. For example, if we examine the EDF server $\sigma(S_6)$ and its clients at time $t = 3$, we see that $S_6$ generates a W.C.E., and $S_7$ takes over its execution. While this has the effect of context switching from $S_2$ to $S_1$ in our schedule of $\mathcal{T}$, the event has no effect on other branches of the tree. On the other hand, because a server inherits its release times from its clients, any release time of a task in $\mathcal{T}$ will become a release time for a client of the terminal unit server, and will cause a scheduler invocation at the top level. Therefore any J.R.E. will cause a rescheduling of the entire tree.


$\sigma(\psi^2(\mathcal{T}))$
$\sigma$
$\psi^2(\mathcal{T})$
$\varphi$
$\sigma(\psi^1(\mathcal{T}))$
$\sigma$
$\psi^1(\mathcal{T})$
$\varphi$
$\sigma(\psi^0(\mathcal{T}))$
$\sigma$
$\psi^0(\mathcal{T})$
EDF$^{(1)}$
$e_1$
$S_{11}^{0.2,\{5\mathbb{N},10\mathbb{N}\}}$
$S_{12}^{0.2,\{10\mathbb{N},15\mathbb{N}\}}$
$S_{13}^{0.6,\{5\mathbb{N}\}}$
$e_2$
$e_3$
$\sigma(S_6)$
$\sigma(S_8)$
$\sigma(S_{10})$
$e_4$
$e_5$
$S_6^{0.4,\{5\mathbb{N}\}}$
$S_7^{0.4,\{10\mathbb{N}\}}$
$S_8^{0.4,\{15\mathbb{N}\}}$
$S_9^{0.4,\{10\mathbb{N}\}}$
$S_{10}^{0.4,\{5\mathbb{N}\}}$
$e_6$
$e_7$
$e_8$
$\sigma(S_1)$
$\sigma(S_2)$
$\sigma(S_3)$
$\sigma(S_4)$
$\sigma(S_5)$
$e_9$
$e_{10}$
$e_{11}$
$S_1^{0.6,\{5\mathbb{N}\}}$
$S_2^{0.6,\{10\mathbb{N}\}}$
$S_3^{0.6,\{15\mathbb{N}\}}$
$S_4^{0.6,\{10\mathbb{N}\}}$
$S_5^{0.6,\{5\mathbb{N}\}}$
$\Sigma_2$
$S_{11}$ $S_{13}$ $S_{12}$ $S_{11}$ $S_{13}$
0 1 2 3 4 5 6 7 8 9 10
$\Sigma_1$
$S_9$ $S_{10}$ $S_{10}$ $S_6$ $S_7$
$S_{10}$ $S_6$ $S_7$ $S_8$
0 1 2 3 4 5 6 7 8 9 10
$\Sigma_0$
$S_1$ $S_5$ $S_4$
$S_2$ $S_1$ $S_1$ $S_5$
$S_3$ $S_2$ $S_1$
0 1 2 3 4 5 6 7 8 9 10

(a) RUN decision tree at time $t = 4$

(b) Run schedule

**Fig. 9** $\mathcal{T} = \{S_1{:}[3/5, 5\mathbb{N}], S_2{:}[3/5, 10\mathbb{N}], S_3{:}[3/5, 15\mathbb{N}], S_4{:}[3/5, 10\mathbb{N}], S_5{:}[3/5, 5\mathbb{N}]\}$. $\Sigma_0$ is the schedule of $\mathcal{T}$ on 3 physical processors. $\Sigma_1$ is the schedule of $\psi(\mathcal{T}) = \{S_6, S_7, S_8, S_9, S_{10}\}$ on 2 virtual processors, and $\Sigma_2$ is the schedule of $\psi^2(\mathcal{T}) = \{S_{11}, S_{12}, S_{13}\}$ on 1 virtual processor

Each child server scheduled by a packed server must keep track of its own workloads and deadlines. These workloads and deadlines are based on the clients of the packed server below it. That is, each server node which is not a task of $\mathcal{T}$ simulates being a task so that its parent node can schedule it along with its siblings in its virtual system. The process of setting deadlines and allocating workloads for virtual server jobs is detailed in Sect. 3.1.

**Algorithm 1**: Outline of the RUN algorithm

```
I. OFF-LINE
    A. Generate a reduction sequence for $\mathcal{T}$
    B. Invert the sequence to form a server tree
    C. For each proper subsystem $\mathcal{T}$' of $\mathcal{T}$
        Define the client/server at each virtual level
II. ON-LINE
Upon a scheduling event:
    A. If the event is a job release event at level 0
        1. Update deadline sets of servers on path up to root
        2. Create jobs for each of these servers accordingly
    B. Apply Rules 1 & 2 to schedule jobs from root to leaves, determining the $m$ jobs to schedule at
    level 0
    C. Assign the $m$ chosen jobs to processors, according to some task-to-processor assignment
    scheme
```

The process described so far from reducing a task set to unit servers, to the scheduling of those tasks with EDF servers and duality, is collectively referred to as the RUN algorithm and is summarized in Algorithm 1. We now finish proving it correct.

**Theorem 5** (RUN correctness) *If $\Gamma$ is a proper set under the reduction sequence $\{\psi^i\}_{i \le p}$, then the RUN algorithm produces a valid schedule $\Sigma$ for $\Gamma$.*

*Proof* Again, let $\Gamma^k = \psi^k(\Gamma)$ and $\Gamma^k_\sigma = \sigma(\Gamma^k)$ with $k < p$. Also, let $\Sigma^k$ and $\Sigma^k_\sigma$ be the schedules generated by RUN for $\Gamma^k$ and $\Gamma^k_\sigma$, respectively.

By Definition 11 of the PACK operation $\sigma$, $\Gamma^k_\sigma$ is the set of servers in charge of scheduling the packing of $\Gamma^k$. Hence, $\rho(\Gamma^k) = \rho(\Gamma^k_\sigma)$. Finally, let $\mu^k = \rho(\Gamma^k) = \rho(\Gamma^k_\sigma)$, which, as seen in the proof of Theorem 4, is always an integer.

We will work inductively to show that schedule validity propagates down the reduction tree, i.e., that the validity of $\Sigma^{k+1}$ implies the validity of $\Sigma^k$.

Suppose that $\Sigma^{k+1}$ is a valid schedule for $\Gamma^{k+1} = \varphi(\Gamma^k_\sigma)$ on $\mu^{k+1}$ processors, where $k + 1 \le p$. Since $k < p$, $\Gamma^k_\sigma$ is not the terminal level set, and so must contain more than one server, as does its equal-sized dual $\Gamma^{k+1}$. Further, since $\Gamma^{k+1}$ is the dual of a packed set, none of these servers can be unit servers, and so $|\Gamma^{k+1}| > \mu^{k+1}$. The conditions of Theorem 2 are satisfied (where $n = |\Gamma^{k+1}|$, $m = \mu^{k+1}$, and $n > m$), so it follows from our assumption of the validity of $\Sigma^{k+1}$ that $\Sigma^k_\sigma = (\Sigma^{k+1})^*$ is a valid schedule for $\Gamma^k_\sigma$ on $\mu^k$ processors.

Now, each aggregated server in $\Gamma^k_\sigma$ meets all of its jobs' deadlines, so according to Theorem 1, every client of each of these servers will meet its deadlines as well. That is, scheduling the servers in $\Gamma^k_\sigma$ correctly ensures that all of their client tasks in $\Gamma^k$ are also scheduled correctly. Thus the validity of $\Sigma^{k+1}$ implies the validity of $\Sigma^k$, as desired.

Since uniprocessor EDF generates a valid schedule $\Sigma^p$ for the clients of the terminal unit server at the final reduction level $p$ (Liu and Layland 1973), it follows inductively that $\Sigma = \Sigma^0$ is valid for $\Gamma$ on $\rho(\Gamma)$ processors. □

## 6 Assessment

### 6.1 RUN implementation

The PACK operation described in Sect. 4.2 is carried out by a standard bin-packing algorithm, where "bins" are servers of capacity one, and items are tasks or other servers whose "sizes" are their rates.

For our implementation of RUN's PACK step, we use best-fit decreasing bin-packing, as it consistently outperforms other bin-packing heuristics (albeit by only a small margin; details may be found in Sect. 6.5). This runs in $O(n \log n)$ time, where $n$ is the number of tasks or servers being packed.

Whenever an intermediate unit server is encountered during the reduction process, we make it a terminal unit server, and isolate its proper subsystem from the rest of the system, as discussed in Sect. 4.3.

At each scheduler invocation, once the set of $m$ running tasks is determined (as in Fig. 9(a)), we use a simple greedy task-to-processor assignment scheme. In three passes through these $m$ tasks, we: first, leave already executing tasks on their current processors; second, assign previously idle tasks to their last-used processor, when its available, to avoid unnecessary migrations; and third, assign remaining tasks to free processors arbitrarily.

Best-fit decreasing bin-packing and EDF are not the only choices for partitioning and uniprocessor scheduling. RUN may be modified so that it reduces to a variety of partitioned scheduling algorithms. Best-fit bin packing can be replaced with any other bin-packing (partitioning) scheme that (i) uses additional "bins" when a proper partitioning onto $m$ processors is not found, and (ii) creates a packed server set. Similarly, any optimal uniprocessor scheduling algorithm can be substituted for EDF. In this way, the RUN scheme can be seen as an extension of different partitioned scheduling algorithms, but one that could, in theory, handle cases when a proper partition on $m$ processors cannot be found.

### 6.2 Reduction complexity

We now observe that the time complexity of a reduction procedure is polynomial and is dominated by the PACK operation. However, as there is no optimality requirement on the (off-line) reduction procedure, any polynomial-time heuristic suffices. There are, for example, linear and log-linear time packing algorithms available (Hochbaum 1997).

**Lemma 4** *If $\Gamma$ is a packed set of at least 2 servers, then $\rho(\Gamma) > |\Gamma|/2$.*

*Proof* Let $n = |\Gamma|$, and let $\mu_i = \rho(S_i)$ for $S_i \in \Gamma$. Since $\Gamma$ is packed, there exists at most one server in $\Gamma$, say $S_n$, such that $\mu_n \leq 1/2$; all others have $\mu_i > 1/2$. Thus, $\sum_{i=1}^{n-2} \mu_i > (n-2)/2$. As $\mu_{n-1} + \mu_n > 1$, it follows that $\rho(\Gamma) = \sum_{i=1}^{n} \mu_i > n/2$. □

**Theorem 6** (Reduction complexity) *RUN's off-line generation of a reduction sequence for n tasks on m processors requires $O(\log m)$ reduction steps and $O(f(n))$ time, where $f(n)$ is the time needed to pack n tasks.*

*Proof* Let $\{\psi^i\}_{i \leq p}$ be a reduction sequence on $\mathcal{T}$, where $p$ is the terminal level described in Theorem 4. Lemma 1 shows that a REDUCE, at worst, reduces the number of servers by about half, so $p = O(\log n)$.

Since constructing the dual of a system primarily requires computing $n$ dual rates, a single REDUCE requires $O(f(n) + n)$ time. The time needed to perform the entire reduction sequence is described by $T(n) \leq T(n/2) + O(f(n) + n)$, which gives $T(n) = O(f(n))$.

Since $\mathcal{T}$ is a full utilization task set, $\rho(\mathcal{T}) = m$. If we let $n' = |\sigma(\mathcal{T})|$, Lemma 4 tells us that $m = \rho(\mathcal{T}) = \rho(\sigma(\mathcal{T})) > n'/2$. But as $\sigma(\mathcal{T})$ is just the one initial packing, it follows that $p$ also is $O(\log n')$, and hence $O(\log m)$. □

### 6.3 On-line complexity

Since the reduction tree is computed off-line, the on-line complexity of RUN can be determined by examining scheduling Rules 1 and 2 and the task-to-processor assignment scheme.

**Theorem 7** (On-line complexity) *Each scheduler invocation of RUN takes $O(n)$ time, for a total of $O(jn\log m)$ scheduling overhead during any time interval when n tasks releasing a total of j jobs are scheduled on m processors.*

*Proof* First, let us count the nodes in the server tree. In practice, $S$ and $\varphi(S)$ may be implemented as a single object / node. There are $n$ leaves, and as many as $n$ servers in $\sigma(\mathcal{T})$. Above that, each level has at most (approximately) half as many nodes as the preceding level. This gives us an approximate node bound of $n + n + n/2 + n/4 + \cdots \leq 3n$.

Next, consider the scheduling process described by Rules 1 and 2. The comparison of clients performed by EDF in Rule 1 does no worse than inspecting each client once. If we assign this cost to the client rather than the server, each node in the tree is inspected at most once per scheduling invocation. Rule 2 is constant time for each node which "dualed". Thus the selection of $m$ tasks to execute is constant time per node, of which there are at most $3n$. The previously described task-to-processor assignment requires 3 passes through a set of $m$ tasks, and so may be done in $O(m) \leq O(n)$ time. Therefore, each scheduler invocation is accomplished in $O(n)$ time.

Since we only invoke the scheduler at W.C.E. or J.R.E. times, any given job (real or virtual) can cause at most two scheduler invocations. The virtual jobs of servers are only released at the release times of their leaf descendants, so a single real job can cause no more than $O(\log m)$ virtual jobs to be released, since there are at most $O(\log m)$ reduction levels (Theorem 6). Each of these resultant virtual jobs will have a different workload, and be running on a different virtual system. So while they will all have the same J.R.E. times, they may all have different W.C.E. times. Thus $j$ real jobs result in no more than $jO(\log m)$ virtual jobs, each of which may cause 1 or 2 distinct $O(n)$ scheduler invocations. Thus a time interval where $j$ real jobs are released will see a total scheduling overhead of $O(jn\log m)$. □

### 6.4 Preemption bounds

We now prove an upper bound on the average number of preemptions per job through a series of lemmas. To do so, we count the preemptions that a job *causes*, rather than the preemptions that a job *suffers*. Thus, while an arbitrarily long job may be preempted arbitrarily many times, the *average* number of preemptions per job is bounded. When a context switch occurs where $A$ begins running and $B$ becomes idle, we say that $A$ *replaces* $B$; if the current job of $B$ still has work remaining, we say that $A$ *preempts* $B$. As before, because all scheduling decisions are made by EDF, we need only consider the preemptions caused by *work complete events* (W.C.E.) and *job release events* (J.R.E.) (which occur concurrently with job deadlines).

**Lemma 5** *Each job from a task or server has exactly one* J.R.E. *and one* W.C.E. *Further, the servers at any one reduction level cannot release more jobs than the original task set over any time interval.*

*Proof* The first claim is obvious and is merely noted for convenience.

Next, since servers inherit deadlines from their clients and jobs are released at deadlines, a server cannot have more deadlines, and hence not release more jobs, than its clients. A server's dual has the same number of jobs as the server itself. Moving inductively up the server tree, it follows that a set of servers at one level cannot have more deadlines, or more job releases, than the leaf level tasks. □

**Lemma 6** *Scheduling a system $\mathcal{T}$ of $n = m + 1$ tasks on $m$ processors with RUN produces an average of no more than one preemption per job.*

*Proof* When $n = m + 1$, there is only one reduction level and no packing; $\mathcal{T}$ is scheduled by applying EDF to its uniprocessor dual system. We claim that dual J.R.E.s cannot cause preemptions in the primal system.

A J.R.E. can only cause a context switch when the arriving job $J_i^*$, say from task $\tau^*$, has an earlier deadline than, and thus replaces, the previously running job. Let $J_i$ be the corresponding job of $\tau^*$'s primal task $\tau$, and recall that $J_i^*.r = J_i.r = J_{i-1}^*.d = J_{i-1}.d$, as illustrated in Fig. 10.

Now let us consider such a context switch, where $J_i^*$ *starts* executing in the dual at its release time $J_i^*.r$. Then in the primal, $\tau$'s previous job $J_{i-1}$ *stops* executing at time $J_{i-1}.d = J_i^*.r$ in the primal. When a job stops executing at its deadline in a valid schedule, it must be the case that it completes its work exactly at its deadline, and stopping a completed job does not count as a preemption. Thus dual J.R.E.s do not cause preemptions in the primal system. By Lemma 5, there can be at most one W.C.E. in the dual, and hence one preemption in the primal, for each job released by a task in $\mathcal{T}$, as desired. □

**Lemma 7** *A context switch at any level of the server tree causes exactly one context switch between two original leaf tasks in $\mathcal{T}$.*

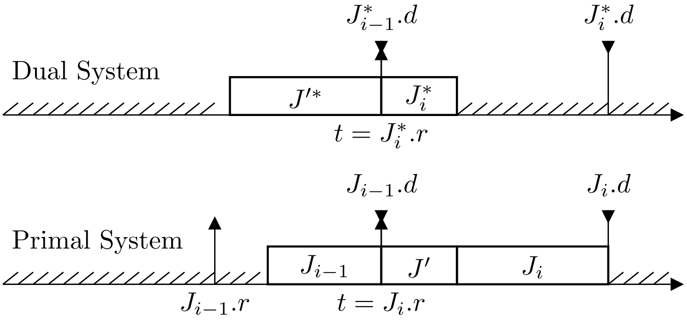


**Fig. 10** In the dual, the arrival of $J^*_i$ preempts $J'^*$. The matching primal event is just the previous job $J_{i-1}$ finishing its work at its deadline, and is not a preemption

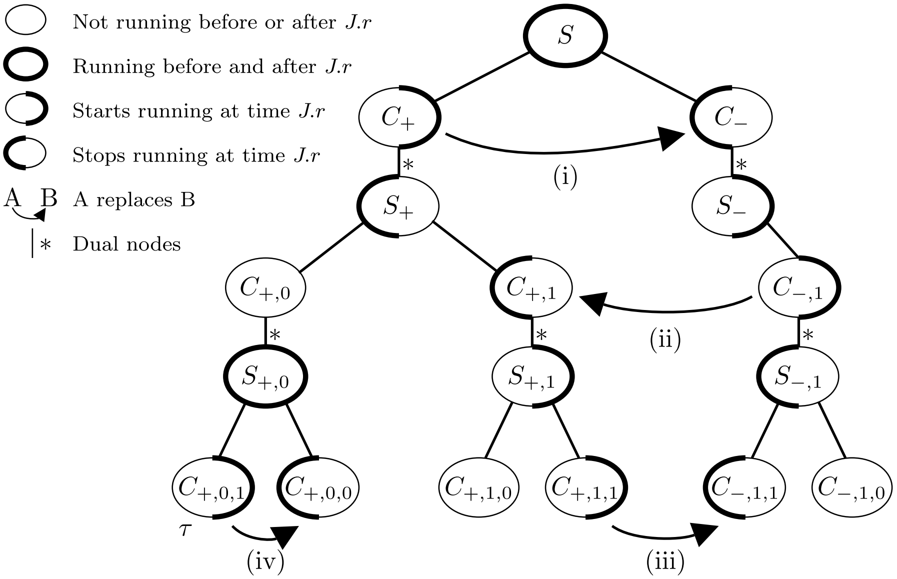


**Fig. 11** Two Preemptions from one JOB RELEASE. In this 3-level (portion of a) server tree, a job release by $\tau$ corresponds to a job release and context switch at the top level (**i**), which propagates down to the right of the tree (**ii, iii**). That same job release by $\tau$ can cause it to preempt (**iv**) another client $C_{+,0,0}$ of its parent server $S_{+,0}$

*Proof* We proceed by induction on the level where the context switch occurs, showing that a context switch at any level of the server tree causes exactly one context switch in the next level below (less reduced than) it.

Consider some tree level where the switch occurs: suppose we have a pair of client nodes (not necessarily of the same server parent) $C_+$ and $C_-$, where $C_+$ replaces $C_-$. All other jobs' "running" statuses at this level are unchanged. Let $S_+$ and $S_-$ be their dual children in the server tree (i.e., $C_+ = S^*_+$ and $C_- = S^*_-$), so by Rule 2, $S_-$ replaces $S_+$ (see Fig. 11 for node relationships).

Now, when $S_+$ was running, it was executing exactly one of its client children, call it $C_{+,1}$; when $S_+$ gets switched off, so does $C_{+,1}$. Similarly, when $S_-$ was off, none of its clients were running; when it gets switched on, exactly one of its clients, say $C_{-,1}$, begins executing. Just as the context switch at the higher (more reduced) level only effects the two servers $C_+$ and $C_-$, so too are these two clients $C_{+,1}$ and $C_{-,1}$ the only clients at this lower level affected by this operation; thus, $C_{-,1}$ must be replacing $C_{+,1}$. So here we see that a context switch at one client level of the tree causes only a single context switch at the next lower client level of the tree (in terms of Fig. 11, (i) causes (ii)). This one context switch propagates down to the leaves, so inductively, a context switch anywhere in the tree causes exactly one context switch in $\mathcal{T}$. □

**Lemma 8** *If RUN requires p reduction levels for a task set $\mathcal{T}$, then any* J.R.E. *by a task $\tau \in \mathcal{T}$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$.*

*Proof* Suppose task $\tau$ releases job $J$ at time $J.r$. This causes a job release at each ancestor server node above $\tau$ in the server tree (i.e., on the path from leaf $\tau$ to the root). We will use Fig. 11 for reference, and note that this only meant to represent the portion of the server tree relevant to our discussion.

Let $S$ be the highest (furthest reduction level) ancestor server of $\tau$ for which this J.R.E. causes a context switch among its clients ($S$ may be the root of the server tree). In such a case, some client of $S$ (call it $C_+$) has a job arrive with an earlier deadline than the currently executing client (call it $C_-$), so $C_+$ preempts $C_-$. As described in the proof of Lemma 7, $C_-$'s dual $S_-$ replaces $C_+$'s dual $S_+$, and this context switch propagates down to a context switch between two tasks in $\mathcal{T}$ (see preemption (iii) in Fig. 11).

However, as no client of $S_+$ remains running at time $J.r$, the arrival of a job for $\tau$'s ancestor $C_{+,0}$ at this level cannot cause a J.R.E. preemption at this time (it may cause a different client of $S_+$ to execute when $S_+$ begins running again, but this context switch will be charged to the event that causes $S_+$ to resume execution). Thus, when an inherited J.R.E. time causes a context switch at one level, it cannot cause a *different* (second) context switch at the next level down. However, it may cause a second context switch two levels down (see preemption (iv)). Figure 11 shows two context switches, (iii) and (iv), in $\mathcal{T}$ that result from a single J.R.E. of $\tau$. One is caused by a job release by $\tau$'s ancestor child of the root, which propagates down to another part of the tree (iii). $\tau$'s parent server is not affected by this, stays running, and allows $\tau$ to preempt its sibling client when its new job arrives (iv).

While $S$ is shown as the root and $\tau$ as a leaf in Fig. 11, this argument would still apply if there were additional nodes above and below those shown, and $\tau$ were a descendant of node $C_{+,0,1}$. If there were additional levels, then $\tau$'s J.R.E. could cause an additional preemption in $\mathcal{T}$ for each two such levels. Thus, if there are $p$ reduction levels (i.e., $p+1$ levels of the server tree), a J.R.E. by some original task $\tau$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. □

**Theorem 8** *Suppose RUN performs p reductions on task set $\mathcal{T}$ in reducing it to a single EDF system. Then RUN will suffer an average of no more than*

$\lceil (3p+1)/2 \rceil = O(\log m)$ *preemptions per job* (*and no more than* 1 *when* $n = m+1$) *when scheduling* $\mathcal{T}$.

*Proof* The $n = m+1$ bound comes from Lemma 6. Otherwise, we use Lemma 5 to count preemptions based on jobs from $\mathcal{T}$ and the two EDF event types. By Lemma 8, a J.R.E. by $\tau \in \mathcal{T}$ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. The context switch that happens at a W.C.E. in $\mathcal{T}$ is, by definition, not a preemption. However, a job of $\tau \in \mathcal{T}$ corresponds to one job released by each of $\tau$'s $p$ ancestors, and each of these $p$ jobs may have a W.C.E. which causes (at most, by Lemma 7) one preemption in $\mathcal{T}$. Thus we have at most $p + \lceil (p+1)/2 \rceil = \lceil (3p+1)/2 \rceil$ preemptions that can be attributed to each job from $\mathcal{T}$, giving our desired result since $p = O(\log m)$ as discussed in Theorem 7. □

In our simulations, we almost never observed a task set that required more than two reductions. For $p = 2$, Theorem 8 gives a bound of 4 preemptions per job. While we never saw more than 3 preemptions per job in our randomly generated task sets, it is possible to do worse.The following 6-task set on 3 processors averages 3.99 preemptions per job, suggesting that our proven bound is tight: $\mathcal{T} = \{(.57, 4000), (.58, 4001), (.59, 4002), (.61, 4003), (.63, 4004), (.02, 3)\}$.

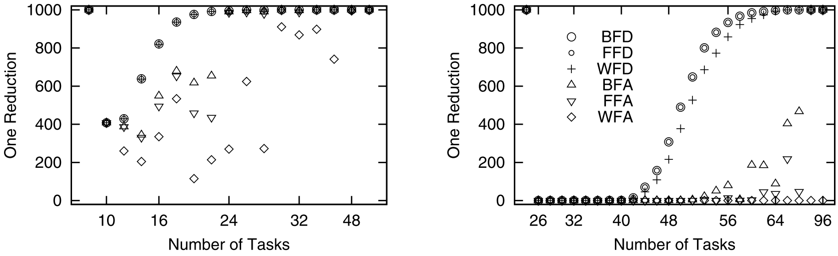
Also, there exist task sets that require more than 2 reductions. A set of only 11 jobs with rates of $7/11$ is sufficient, with a primal reduction sequence:

$$\left\{(11)\,\frac{7}{11}\right\} \rightarrow \left\{(5)\,\frac{8}{11}, \frac{4}{11}\right\} \rightarrow \left\{\frac{10}{11}, \frac{9}{11}, \frac{3}{11}\right\} \rightarrow \{1\}.$$

Such constructions require narrowly constrained rates and randomly generated task sets requiring 3 or more reductions are rare. A 3-reduction task set was observed on 18 processors, and a 4-reduction set appeared on 24 processors, but even with 100 processors and hundreds of tasks, 3- and 4-reduction sets occur in less than 1 in 600 of the random task sets generated.

### 6.5 Heuristics

We chose best-fit decreasing as our bin-packing subroutine based on simulations involving multiple bin-packing heuristics (methodology of simulation will be discussed in the next section). By far the strongest indicator of performance, measured in terms of preemptions per job, is the number of reduction levels required by a given task set. And as Table 3 shows, the number of reductions is a function of the particular packing used. In all of our various simulations on randomly generated task sets, we only encountered task sets that required 0, 1, or 2 reduction levels, and 0 reduction levels were only found at lower total utilizations. We have tested the best-fit, first-fit and worst-fit bin-packing heuristics with sizes (rates) arranged in both arbitrary and decreasing order. Figure 12 shows the number of task sets out of 1000 simulations which require only one reduction level, as a function of the number of tasks. As fewer reduction levels are preferred, we see that packing tasks in decreasing order always outperforms arbitrary order. Although the four decreasing heuristics we tried



(a) 8 processors

(b) 24 processors

**Fig. 12** Number of task sets out of 1000 requiring 1 reduction for RUN simulations on 8 and 24 processors at full utilization. Any task set not requiring 1 reduction required only 2

performed similarly, best-fit was consistently the best, and hence was our choice for RUN's implementation.

Since we observed that all decreasing bin-packing heuristics perform similarly, we concluded that, so long as tasks are packed in decreasing rate order, it does not matter much which bin a task is placed in, so long as it fits. With this in mind, we tested a different bin-packing criteria, where bins were selected based on compatibility of task periods. As with other decreasing heuristics, tasks are placed in bins one at a time, in decreasing rate order, with a new bin created whenever a task doesn't fit in any existing bin. When a task would fit in multiple bins, we select the bin where the addition of the task caused the smallest increase to the *Least Common Multiple* (LCM) of the periods of tasks already in that bin. For this purpose, the "period" of an aggregated server was taken to be the LCM of the "periods" of its clients. The benefit of this approach is that, when tasks of compatible (i.e., large common divisor) periods are grouped together in servers, those servers have fewer job releases, and consequently cause fewer scheduler invocations and preemptions. As a simple example, imagine we have server bins that currently have periods of 10 and 19, and we wish to add a task $\tau$ with period 20 to one of these. If we add it to the period 19 bin, then nearly every job release of $\tau$ is going to cause a distinct job release for its server (multiples of 19 and 20 rarely coincide). On the other hand, if we add it to the period 10 bin, then $\tau$ causes *no* additional job releases for this server (every multiple of 20 is already a multiple of 10).

Figure 13 shows average migrations- and preemptions-per-job for a varying number of tasks simulated on 16 processors. As in Fig. 12, best-fit slightly outperformed our new "LCM-fit" heuristic in terms of task sets requiring only one reduction level. However, when task set schedules were simulated, the LCM-fit packer suffered 4–5 % fewer preemptions and migrations per job than any of the other bin-packers. This is attributable to the reduction in server jobs that comes from grouping tasks of compatible periods onto the same server. In spite of this slightly improved performance, we chose to use the best-fit heuristic for our primary simulations. The LCM-fit bin packer is complex, and its particular benefit is heavily dependent on our choice of randomly generating integral periods in the range [5, 100]. We include these results merely to demonstrate that there is potential benefit in grouping tasks according to periods, and that this benefit could be significant in environments where some tasks have strongly compatible periods.

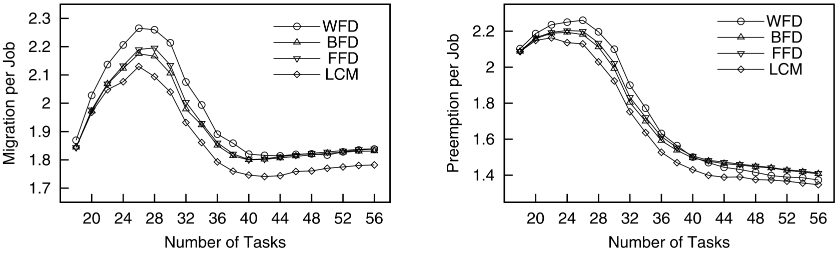


**Fig. 13** Migrations- and preemptions-per-job by RUN using 4 different bin-packing heuristics. Simulations were run on $m = 16$ processors for $n = 18, \ldots, 56$ tasks at 100 % utilization

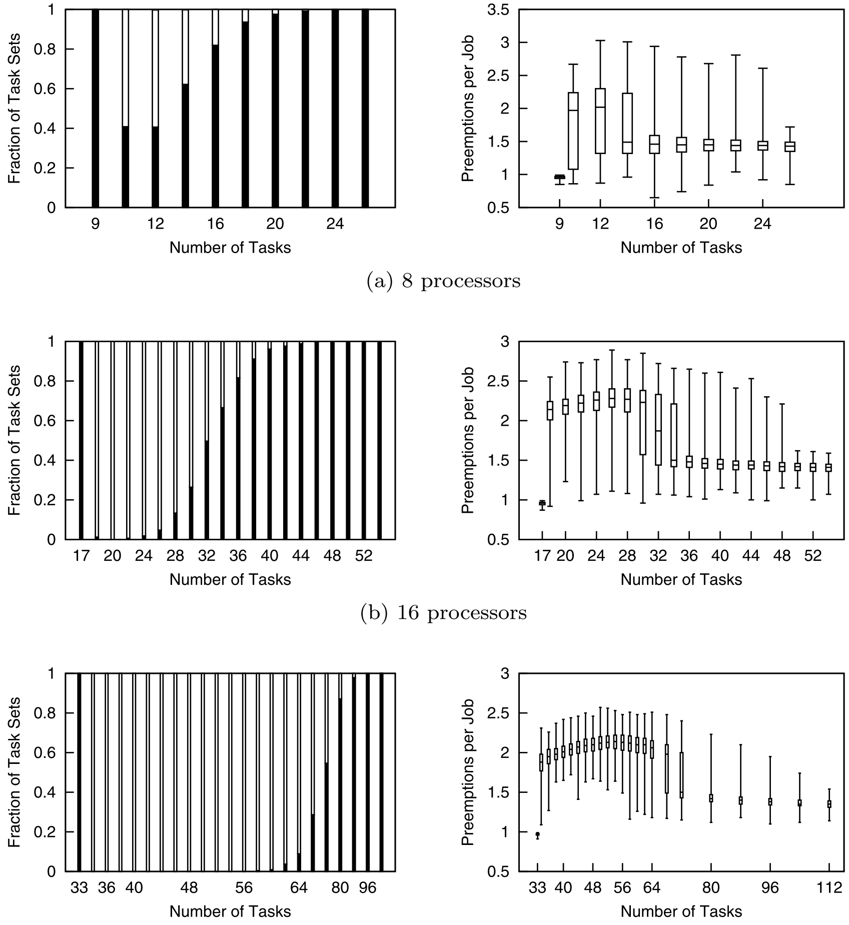
Recall from Sect. 4 that duality is only defined for task sets with 100 % utilization. However, task sets will typically fall short of 100 % utilization. In such cases, this deficit may be made up by the addition of any collection of dummy tasks whose rates add up to $m - \rho(\mathcal{T})$, so long as none of these dummy tasks has rate $\rho > 1$. If done well, this may improve performance. To this end, we introduce the *slack packing* heuristic to distribute a task system's *slack* (defined as $m - \rho(\mathcal{T})$) among the aggregated servers at the end of the initial PACK step. Servers are filled to become unit servers, and then isolated from the system. The result is that some or all processors are assigned only non-migrating tasks and behave as they would in a partitioned schedule.

For example, suppose that the task set from Fig. 9 runs on four processors instead of three. The initial PACK can only place one 0.6 utilization task per server. From the 1 unit of slack provided by our fourth processor, we create a dummy task $S_1^d$ with $\rho(S_1^d) = 0.4$ (and arbitrarily large deadline), pack it with $S_1$ to get a unit server and give it its own processor. Similarly, $S_2$ also gets a dedicated processor. The remaining 0.2 units of slack are put into a third dummy task, which is scheduled along with $S_3$, $S_4$, and $S_5$ on the remaining two processors in the usual fashion. But now $S_1$ and $S_2$ never need preempt or migrate, so the schedule is more efficient. With 5 processors, this approach yields a fully partitioned system, where each task has its own processor. With low enough utilization, the first PACK usually results in $m$ or fewer servers. In these cases, slack packing gracefully reduces RUN to Partitioned EDF.

## 6.6 Simulation

We have evaluated RUN via extensive simulation using task sets generated for various levels of $n$ tasks, $m$ processors, and total utilization $\rho(\mathcal{T})$. Task rates were generated in the range of [.01, .99] following the Emberson task generation procedure (Emberson et al. 2010) based on the random fixed-sum vector generator of Stafford (2006). Task periods were drawn independently from a uniform integer distribution in the range [5, 100] and simulations were run for 1000 time units. Values reported for migrations and preemptions are *per job* averages, that is, total counts were divided by the number of jobs released during the simulation, averaged over all task sets. For each data point shown, 1000 task sets were generated.

For direct evaluation, we considered systems of $m = 8$, 16 and 32 processors. For each of these, we examined sets of $n$ tasks for values of $n$ in the approximate range



(a) 8 processors

(b) 16 processors

(c) 32 processors

**Fig. 14** Fraction of task sets requiring 1 (filled box) and 2 (empty box) reduction levels; Distributions of the average number of preemptions per job, their quartiles, and their minimum and maximum values. All RUN simulations on 8, 16 and 32 processor systems are at full utilization

of $m + 1$ through $3m$. For each $(m, n)$ pair, we simulated 1000 randomly generated task sets at 100 % processor utilization. We measured the number of reduction levels and the number of preemption points. Job completion is not considered a preemption point.

The left plots of Fig. 14 show the number of reduction levels; none of the task sets generated require more than two reductions. For $m = n + 1$ (9, 17 and 33) tasks, only one level is necessary, as seen in Fig. 2, and implied by Theorem 2. For 8, 16 and 32 processors, we observe that one or two levels are needed for $n \in [10, 22]$, $n \in [18, 44]$ and $n \in [34, 95]$, and only one level is ever needed for $n > 22$, $n > 44$ and $n > 96$, respectively. With low average task rates, the first PACK gives servers with rates close to 1; the very small dual rates then sum to 1, yielding the terminal level.

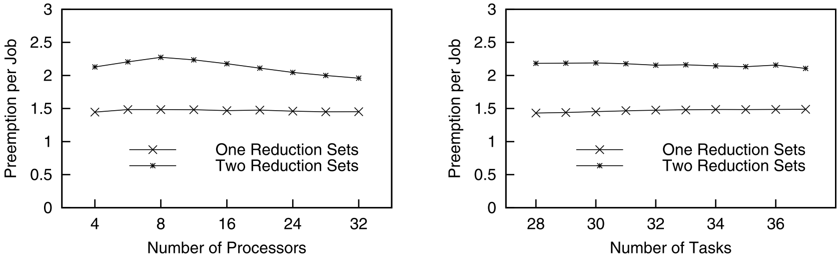


**Fig. 15** Preemptions-per-job by RUN on task sets requiring 1 and 2 reduction levels. In the first plot, $m = 4, \dots, 32$ and $n = 2m$; in the second plot, $m = 16$ and $n = 28, \dots, 37$; both show task sets with 100 % utilization

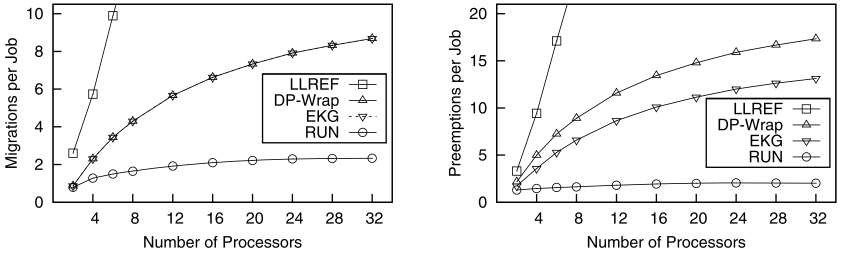


**Fig. 16** Migrations- and preemptions-per-job by LLREF, DP-Wrap, EKG, and RUN as number of processors $m$ varies from 2 to 32, with full utilization and $n = 2m$ tasks. Note: DP-Wrap and EKG have the same migration curves

The box-plots in the right column of Fig. 14 show the distribution of preemptions as a function of the number of tasks. We see a strong correlation between the number of preemptions and number of reduction levels; where there is mostly only one reduction level, preemptions per job is largely independent of the size of the task set. Indeed, for $n \geq 16$, $n \geq 36$ and $n \geq 80$, the median preemption count stays nearly constant just below 1.5.

This correlation between preemptions and reduction levels is shown explicitly in Fig. 15. In the two plots, we vary number of processors and number of tasks, while keeping the number of tasks in a range where both 1- and 2-reduction task sets are common. We observe that average preemptions per job is nearly constant as processors and tasks vary, and is almost entirely a function of whether a task set requires 1 or 2 reduction levels. 1 reduction task sets average about 1.46 preemptions per job, while 2 reduction task sets average about 2.15 preemptions per job. Although the range bars in Fig. 14 show considerable variance between task sets, no task set generated for these simulations ever incurred more than 3 average preemptions per job.

Next, we ran comparison simulations against other optimal algorithms. In Fig. 16, we count migrations and preemptions made by RUN, LLREF (Cho et al. 2006), EKG (Andersson and Tovar 2006) and DP-Wrap (Levin et al. 2010) (with these last two employing the simple *mirroring* heuristic) while increasing processor count from 2 to 32. Most of LLREF's results are not shown to preserve the scale of the rest of

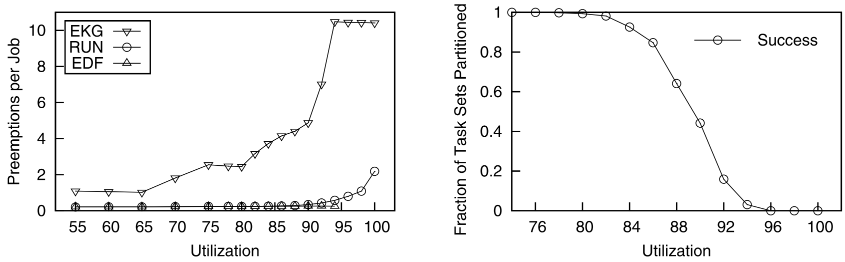


**Fig. 17** Preemptions per job for EKG, RUN, and Partitioned EDF as utilization varies from 55 to 100 %, with 24 tasks on 16 processors; Partitioning success rate for best-fit bin packing under the same conditions

the data. Whereas the performance of LLREF, EKG and DP-Wrap get substantially worse as $m$ increases, the overhead for RUN quickly levels off, showing that RUN scales quite well with system size.

Finally, we simulated EKG, RUN, and Partitioned EDF at lower task set utilizations (LLREF and DP-Wrap were excluded, as they consistently perform worse than EKG). Because 100 % utilization is unlikely in practice, and because EKG is optimized for utilizations in the 50–75 % range, we felt these results to be of particular interest. For RUN, we employed the slack-packing heuristic. Because this often reduces RUN to Partitioned EDF for lower utilization task sets, we include Partitioned EDF for comparison in Fig. 17's preemptions per job plot. Values for Partitioned EDF are only averaged over task sets where a successful partition occurs, and so stop at 94 % utilization. The second plot shows the fraction of task sets that achieve successful partition onto $m$ processors, and consequently, where RUN reduces to Partitioned EDF.

With its few migrations and preemptions at full utilization, its efficient scaling with increased task and processor counts, and its frequent reduction to Partitioned EDF on lower utilization task sets, RUN represents a substantial performance improvement in the field of optimal schedulers.

## 7 Related work

Real-time multiprocessor schedulers are categorized by how and if migration is used. *Partitioned* approaches are simpler because they disallow migration, but can leave significant unused processor capacity. Our work focuses on the higher utilizations allowed by *global* algorithms.

There have been several optimal global scheduling approaches for the PPIID model. If all tasks share the same deadline, the system can be optimally scheduled with very low implementation cost (McNaughton 1959). Optimality was first achieved without this restriction by *pfair* (Baruah et al. 1996), which keeps execution times close to their proportional allotments (*fluid rate curves*). With scheduler invocations at every multiple of some discrete time quantum, preemptions and migrations are high.

More recent optimal schedulers (Andersson and Tovar 2006; Cho et al. 2006; Funk et al. 2011; Levin et al. 2010; Zhu et al. 2003) also rely on proportional fairness, but only enforce it at task deadlines. Time is partitioned into slices based on the deadlines of all tasks in the system and workloads are assigned in each slice proportional to task rates. This creates an environment where all deadlines are equal, greatly simplifying the scheduling problem. Some of these approaches (Cho et al. 2006; Funaoka et al. 2008) envision the time slices as *T-L Planes*, with work remaining curves constrained by a triangular feasible region. Others have extended these approaches to more general problem models (Funk 2010; Levin et al. 2010). Regardless of the specifics, $O(n)$ or $O(m)$ scheduler invocations are necessary within each time slice, again leading to a large number of preemptions and migrations.

In a recent work (Nelissen et al. 2011), a new algorithm called U-EDF (**U**nfair scheduling algorithm based on **EDF**) has been proposed. U-EDF uses a DP-Fair algorithm (Levin et al. 2010), but relaxes the proportionate fairness assumption in order to decrease the need for preemptions and migrations. While not proven to be optimal, U-EDF correctly scheduled more than thousand randomly generated task sets. In all those experiments, U-EDF significantly reduced the average number of preemptions and migrations per job when compared to previous solutions.

Other recent works have used the *semi-partitioning* approach to limit migrations (Andersson and Tovar 2006; Andersson et al. 2008; Easwaran et al. 2009; Kato et al. 2009; Massa and Lima 2010). Under this scheme, some tasks are allocated off-line to processors, much like in the partitioned approach, while other tasks migrate, the specifics of which are handled at run-time. These approaches present a trade-off between implementation overhead and achievable utilization; optimality may be achieved at the cost of high migration overhead.

RUN employs a semi-partitioned approach, but partitions tasks among servers rather than processors. RUN also introduces the *new deadline sharing approach*: each server generates a job between consecutive deadlines of any client tasks, and that job is assigned a workload proportional to the server's rate. The client jobs of a server collectively perform a proportionally "fair" amount of work between any two client deadlines, but such deadlines do not demand fairness among the individual client tasks and tasks in different branches of the server tree may have little influence on each others' scheduling. This is in stark contrast to previous optimal algorithms, where every unique system deadline imposes a new time slice and such slices cause preemptions for many or all tasks. The limited isolation of groups of tasks provided by server partitioning and the reduced context switching imposed by minimal proportional fairness make RUN significantly more efficient than previous optimal algorithms.

Other related work may be found on the topics of duality and servers. Dual systems and their application in scheduling $m+1$ tasks on $m$ fully utilized processors (Levin et al. 2009) is generalized by our approach. The concept of task servers has been extensively used to provide a mechanism to schedule soft real-time tasks (Liu 2000), for which timing attributes like period or execution time are not known *a priori*. There are server mechanisms for uniprocessor systems which share some similarities with one presented here (Deng et al. 1997; Spuri and Buttazzo 1996). Other server mechanisms have been designed for multiprocessor systems, e.g., (Andersson and Tovar 2006; Andersson et al. 2008; Moir and Ramamurthy 1999). Unlike these previous approaches, RUN uses the server abstraction as part of a reduction scheme which hides the complexities of multiprocessor scheduling, simplifying all on-line scheduling decisions to simple uniprocessor EDF scheduling.

## 8 Conclusion

We have presented the optimal RUN multiprocessor real-time scheduling algorithm. RUN transforms the multiprocessor scheduling problem into an equivalent set of uniprocessor problems. Theory and simulation show that only a few preemption points per job are generated on average, allowing RUN to significantly outperform prior optimal algorithms. RUN reduces to the more efficient partitioned approach of Partitioned EDF whenever best-fit bin packing finds a proper partition, and scales well as the number of tasks and processors increase.

These results have both practical and theoretical implications. The overhead of RUN is low enough to justify implementation on actual multiprocessor architectures. At present, our approach only works for fixed-rate task sets with implicit deadlines. Theoretical challenges include extending the model to more general problem domains such as sporadic tasks with constrained deadlines. The use of uniprocessor scheduling to solve the multiprocessor problem raises interesting questions in the analysis of fault tolerance, energy consumption and adaptability. We believe that this novel approach to optimal scheduling introduces a fertile field of research to explore and further build upon.

**Acknowledgements** This work was funded by CAPES (Brazil), CNPq (Brazil), NSF (USA) and Los Alamos National Laboratory (USA).

## References

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: IEEE embedded and real-time computing systems and applications (RTCSA), pp 322–334

Andersson B, Bletsas K, Baruah SK (2008) Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: IEEE real-time systems symposium (RTSS), pp 385–394

Baruah SK (2001) Scheduling periodic tasks on uniform multiprocessors. Inf Process Lett 80(2):97–104

Baruah SK, Goossens J (2004) Scheduling real-time tasks: algorithms and complexity. In: Leung JYT (ed) Handbook of scheduling: algorithms, models, and performance analysis. Chapman Hall/CRC Press, London/Boca Raton

Baruah SK, Mok AK, Rosier LE (1990) Preemptively scheduling hard-real-time sporadic tasks on one processor. In: IEEE real-time systems symposium (RTSS), pp 182–190

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1993) Proportionate progress: a notion of fairness in resource allocation. In: ACM symposium on the theory of computing (STOC). ACM, New York, pp 345–354

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15(6):600–625

Cho H, Ravindran B, Jensen ED (2006) An optimal real-time scheduling algorithm for multiprocessors. In: IEEE real-time systems symposium (RTSS), pp 101–110

Deng Z, Liu JWS, Sun J (1997) A scheme for scheduling hard real-time applications in open system environment. In: Euromicro conference on real-time systems (ECRTS), pp 191–199

Easwaran A, Shin I, Lee I (2009) Optimal virtual cluster-based multiprocessor scheduling. Real-Time Syst 43(1):25–59

Emberson P, Stafford R, Davis RI (2010) Techniques for the synthesis of multiprocessor tasksets. In: Workshop on analysis tools and methodologies for embedded and real-time systems (WATERS), pp 6–11. http://retis.sssup.it/waters2010/data/taskgen-0.1.tar.gz

Funaoka K, Kato S, Yamasaki N (2008) Work-conserving optimal real-time scheduling on multiprocessors. In: Euromicro conference on real-time systems (ECRTS), pp 13–22

Funk S (2010) LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. Real-Time Syst 46(3):332–359

Funk S, Levin G, Sadowski C, Pye I, Brandt S (2011) DP-FAIR: a unifying theory for optimal hard real-time multiprocessor scheduling. Real-Time Syst 47(5):389–429

Hochbaum DS (ed) (1997) Approximation algorithms for NP-hard problems. PWS, Boston

Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Euromicro conference on real-time systems (ECRTS), pp 249–258

Koren G, Amir A, Dar E (1998) The power of migration in multi-processor scheduling of real-time systems. In: ACM-SIAM symposium on discrete algorithms (SODA), pp 226–235

Levin G, Sadowski C, Pye I, Brandt S (2009) SnS: a simple model for understanding optimal hard real-time multi-processor scheduling. Tech. Rep. UCSC-SOE-11-09, Univ. of California, Santa Cruz

Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In: Euromicro conference on real-time systems (ECRTS), pp 3–13

Liu CL (1969) Scheduling algorithms for multiprogram in a hard real-time environment. JPL Space Programs Summary II:37–60

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogram in a hard real-time environment. J ACM 20(1):46–61

Liu JWS (2000) Real-time systems. Prentice-Hall, New York

Massa E, Lima G (2010) A bandwidth reservation strategy for multiprocessor real-time scheduling. In: IEEE real-time and embedded technology and applications symposium (RTAS), pp 175–183

McNaughton R (1959) Scheduling with deadlines and loss functions. Manag Sci 6(1):1–12

Moir M, Ramamurthy S (1999) Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In: IEEE real-time systems symposium (RTSS), pp 294–303

Nelissen G, Berten V, Goossens J, Milojevic D (2011) Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In: IEEE embedded and real-time computing systems and applications (RTCSA), pp 15–24

Regnier P, Lima G, Massa E, Brandt S LG (2011) Run: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: IEEE real-time systems symposium (RTSS), pp 104–115

Spuri M, Buttazzo G (1996) Scheduling aperiodic tasks in dynamic priority systems. Real-Time Syst 10(2):179–210

Stafford R (2006) Random vectors with fixed sum. http://www.mathworks.com/matlabcentral/fileexchange/9700

Zhu D, Mossé D, Melhem R (2003) Multiple-resource periodic scheduling problem: how much fairness is necessary? In: IEEE real-time systems symposium (RTSS), pp 142–151

Zhu D, Qi X, Mossé D, Melhem R (2011) An optimal boundary fair scheduling algorithm for multiprocessor real-time systems. J Parallel Distrib Comput 71(10):1411–1425

**Paul Regnier** received a B.Physics degree from the University Paris XI, France, in 1990 and the M.Sc. degree and the Ph.D. degree in Computer Science from the Federal University of Bahia (UFBa), Brazil, in 2008 and 2012, respectively. He is currently a temporary professor of Numerical Calculus and Computer Science at UFBa. His main research interests are in the area of real-time systems, covering topics such as scheduling, operating systems, communication networks, formal specification and distributed systems. He recently coauthored the paper: "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor.", which received the best paper award at RTSS 2011.

**George Lima** received the B.Sc. degree from Federal University of Bahia, Brazil, in 1993, the M.Sc. degree from the State University of Campinas, Brazil, in 1996, and the Ph.D. degree from the University of York in 2003, all degrees in Computer Science. Since 2004, he has taken a permanent position at Federal University of Bahia as Professor, where he was the Head of Department from 2006 to 2008 and got involved in several relevant administrative activities, helping setting up the Computer Science Graduate programs. His main research interests are in the area of real-time systems, covering topics such as scheduling, fault tolerance, operating systems, communication networks and, distributed systems.

**Ernesto Massa** received his B.Sc. degree in Computer Science and his M.Sc. degree in Mathematics from Federal University of Bahia in 1991 and 2004, respectively. Since 2008 he has been a Ph.D. student in the Computer Science Department at the same university. During his master Ernesto worked on optimization and scheduling problems. His current research focus is on real-time scheduling theory. Since 2010 Ernesto has taken a part-time permanent position at State University of Bahia.

**Greg Levin** holds a Ph.D. in Mathematical Sciences from the Johns Hopkins University, where he wrote his dissertation on Fractional Graph Theory. He was a visiting professor of mathematics at Harvey Mudd College for four years, and a software developer for five. He is currently pursuing a Ph.D. in Computer Science at U.C. Santa Cruz. He is working with Scott Brandt in the Systems Research Lab, where he studies Real-Time Multiprocessor Scheduling.

**Scott Brandt** received the B.Math. degree and the M.S. degree in Computer Science from the University of Minnesota in 1987 and 1993, respectively, and the Ph.D. degree in Computer Science from the University of Colorado at Boulder in 1999. He is currently Professor of Computer Science at the University of California, Santa Cruz (UCSC), and Director of the UCSC Systems Research Laboratory and the UCSC/Los Alamos National Laboratory Institute for Scalable Scientific Data Management (ISSDM). His research interests are broadly in the area of computer systems and his research ranges from scalable, high-performance distributed storage to real-time CPU scheduling.