RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor

Paul Regnier[†], George Lima[†], Ernesto Massa[‡]
[†]Federal University of Bahia, [‡]State University of Bahia
Salvador, Bahia, Brasil
Email: {pregnier, gmlima, ernestomassa}@ufba.br

Greg Levin, Scott Brandt
University of California
Santa Cruz, USA
{glevin,sbrandt}@soe.ucsc.edu

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 ($\leqslant 3$ 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

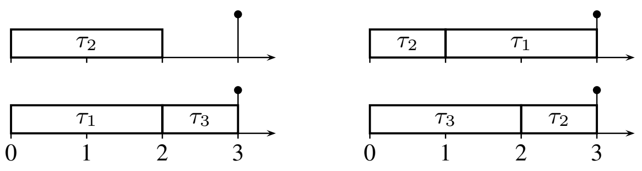
I. Introduction

A. Motivation

Optimal multiprocessor real-time scheduling is challenging. Several solutions have recently been presented, most based on periodic-preemptive-independent (PPID) tasks with implicit deadlines. We address a generalization of this with the goal of finding an efficient valid schedule (*i.e.,* meeting all deadlines).

Multiprocessor scheduling is often achieved via *partitioned* approaches in which tasks are statically assigned to processors, guaranteeing only 50% utilization in the worst case [1]. *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, τ_1, τ_2 and τ_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 Figure 1 (a)). If tasks may migrate, all three tasks can be completed in the time available (Figure 1(b)). This is a simple example of McNaughton's wrap-around algorithm [2], 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 [3]–[7], all relying on some version of *proportional fairness* and, like McNaughton's algorithm, all relying upon the simplicity of scheduling when deadlines are equal. Most enforce deadline equality by proportionally subdividing workloads and imposing the deadlines



(a) τ_3 misses its deadline (b) Correct schedule with migration

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

of each task on all other tasks [7]. This causes many tasks to execute between every two consecutive system deadlines, leading to excessive context switching and migration overhead. We present RUN (Reduction to UNiprocessor), a new optimal algorithm based on a weak version of proportional fairness with significantly lower overhead.

B. Approach

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. Tasks are independent and fully preemptable with implicit deadlines, *i.e.,* the deadline of a task's job is the release time of its next job. We do not assume that tasks are periodic. Instead, tasks have a fixed rate and a job of a task with rate $\mu \leqslant 1$ requires $\mu(d - r)$ execution time. This is equivalent to the PPID model for periodic tasks.

RUN produces a valid multiprocessor schedule for a set of fixed-rate tasks with few preemptions per job, regardless of the number of processors, tasks, or jobs, averaging less than 3 preemptions per job in all of our simulations. RUN works by 1) reducing the real-time multiprocessor scheduling problem to an equivalent set of easily solved uniprocessor scheduling problems through two operations: DUAL and 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 Figure 1, RUN creates a "dual" task set $\{\tau_1^*, \tau_2^*, \tau_3^*\}$ where each dual task τ_i^* has the same deadline as τ_i and a complementary workload of $3 - 2 = 1$. The dual τ_i^* of task τ_i represents τ_i's idle time; τ_i^* executes exactly when τ_i is idle, and vice versa (see Figure 2). These three dual tasks can be scheduled to execute on a single dual processor before their deadlines at time 3. A schedule for the original task set is obtained by blocking τ_i whenever τ_i^* executes in the dual schedule.

This example required a single reduction. In general, RUN performs a sequence of transformations, iteratively reducing the number of processors until one or more uniprocessor systems are derived. RUN uses Earliest Deadline First (EDF) to schedule the uniprocessor problems and iteratively unpacks the solutions as above to yield a valid schedule for the original multiprocessor system. 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 up, starting with the uniprocessor schedule and working backwards to the multiprocessor schedule. Moreover, in each schedule there are

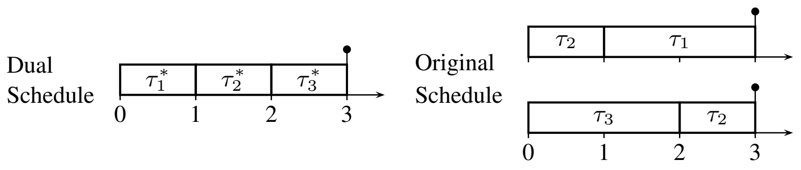


Fig. 2. Scheduling equivalence of τ_1, τ_2, τ_3 on two processors and τ_1^*, τ_2^* τ_3^* on one processor.

always exactly as many tasks (or servers) selected for execution as there are processors.

The remainder of this paper is as follows: Section II describes our system model and notation (summarized in Table I). Section III introduces servers, which aggregate sets of tasks and other servers into schedulable entities. Section IV describes and proves the correctness of the the RUN scheduling algorithm's transformation from multiprocessor to uniprocessor and back. Section V provides implementation details, proofs of performance and preemption bounds, and simulation results comparing RUN with previous solutions. Section VI briefly surveys related work and Section VII presents our conclusions.

II. System Model and Notation

A. Fixed-Rate Tasks

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

Definition II.1 (Job)**.** *A real-time* job *J is a finite sequence of instructions with a release time $J.r$, an execution time $J.c$, and a deadline $J.d$.*

To represent possibly non-periodic execution, a *task* is defined in terms of a constant execution rate $\mu \leqslant 1$.

Definition II.2 (Fixed-Rate Task)**.** *Let $\mu \leqslant 1$ be a positive real number and D a countably infinite set of positive integers. The* fixed-rate task *τ with rate μ and deadlines D, denoted $\tau{:}(\mu, D)$, releases an infinite sequence of jobs satisfying the following properties: (i) a job J of τ is released at time t if and only if $t = 0$ or $t \in D$; (ii) if J is released at time $J.r$, then $J.d = \min_t\{t \in D, t > J.r\}$; and (iii) $J.c = \mu(J.d - J.r)$.*

A fixed-rate task τ has rate $\mathrm{R}(\tau)$ and deadline set $\mathrm{D}(\tau)$. All tasks in this paper are fixed-rate tasks, referred to henceforth simply as "tasks". As a simple example, a periodic task τ with start time of $t = 0$, period T, and execution time C is a fixed-rate task with rate $\mathrm{R}(\tau) = C/T$, and deadlines $\mathrm{D}(\tau) = \{jT, j \in \mathbb{N}^*\}$. We use the more general model of a fixed-rate task because it can also represent groups of tasks, with rate equal to the sum of the group's rates and deadlines equal to the union of the group's deadlines.

B. Fully Utilized System

A system is fully utilized if the sum of the rates of the tasks equals the number of processors. We assume full system utilization, since idle tasks may

TABLE I
SUMMARY OF NOTATION

τ; S; J	Fixed-rate task; Server; Job
$\tau : (\mu, D)$	Task with rate μ and deadline set D
$J.r$; $J.c$; $J.d$	Release time; Execution time; Deadline of J
$e(J, t)$	Work remaining for job J at time t
$\mathcal{T}$; Γ	Set of tasks; Set of servers
$\mathrm{R}(\tau)$; $\mathrm{R}(\Gamma)$	Rate of task τ; Rate of task set Γ
$\mathrm{D}(\tau)$	Set of deadlines of task τ
Σ	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
$\mathrm{ser}(\Gamma)$	Server of the set of tasks (servers) Γ
τ^*, $\varphi(\tau)$	Dual task of task τ, DUAL operator
$\pi(\Gamma)$	Partition of Γ according to packing π
$\sigma(S)$	Server of S given by PACK operation
$\psi = \varphi \circ \sigma$	REDUCE operator

be inserted as needed to fill in slack. Similarly, if a job does not require its full worst-case execution time estimate, we may fill in the difference with forced idle time. If we wish a task to have an initial job release at some time $s > 0$, we may add a dummy job J_0 with release time 0 and deadline s. Hence, without loss of generality, we assume that m jobs are executing at all times in any correct schedule.

C. 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 take zero time. In an actual system, measured preemption and migration overheads can be accommodated by adjusting our target utilizations.

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. If $c(J, t)$ is the amount of time that J has executed as of time t, then $e(J, t) = J.c - c(J, t)$.

Definition II.3 (Schedule)**.** *For a set of jobs (or tasks) $\mathcal{J}$ on a platform of m identical processors, a* schedule Σ *is a function from the set of all non-negative times t onto the set of all subsets of $\mathcal{J}$ such that (i) $|\Sigma(t)| \leqslant m$ for all t, and (ii) if $J \in \Sigma(t)$, then $J.r \leqslant t$, $e(J, t) > 0$ and J executes at t. Thus, $\Sigma(t)$ is the set of jobs executing at time t.*

That is, only execute jobs which have been released and have work remaining.

Once we have demonstrated that our algorithm produces valid schedules, we will explain how to do processor assignment (see Section V).

Definition II.4 (Valid Schedule)**.** *A schedule* Σ *of a job set* $\mathcal{J}$ *is* valid *if all jobs in* $\mathcal{J}$ *finish by their deadlines.*

For a full utilization task set, a valid schedule must have $|\Sigma(t)| = m$ for all times t. A set of jobs or tasks is *feasible* if a valid schedule for it exists. Each fixed-rate task has at most one ready job at any time in a valid schedule. A scheduling algorithm is *optimal* if it finds a valid schedule for all feasible sets of fixed-rate tasks. We now describe the construction of our optimal RUN algorithm.

III. 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. When a server is running, the processor time is used by one of its clients. Server clients are scheduled via an internal scheduling mechanism.

The rate of a server is never greater than one, so this section focuses only on uniprocessor systems. We precisely define the concept of servers (see Section III-A) and show how they correctly schedule the tasks associated with them (Section III-B). We return to multiprocessors in the following section.

A. Server model and notations

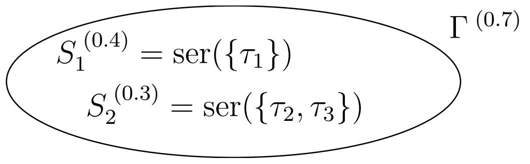
A server for a set of tasks is defined as follows:

Definition III.1 (Server/Client)**.** *Let* $\mathcal{T}$ *be a set of tasks with total rate given by* $\mathrm{R}(\mathcal{T}) = \sum_{\tau\in\mathcal{T}} \mathrm{R}(\tau) \leqslant 1$*. A* server S *for* $\mathcal{T}$*, denoted* $\mathrm{ser}(\mathcal{T})$*, is a virtual task with rate* $\mathrm{R}(\mathcal{T})$ *and deadlines* $\mathrm{D}(S) = \cup_{\tau\in\mathcal{T}} \mathrm{D}(\tau)$*.* $\mathcal{T}$ *is the set of* S*'s* clients, *which is denoted* $\mathrm{cli}(S)$*.* S *is equipped with a scheduling policy to schedule the jobs of its clients.*

Client/server relationships are statically determined prior to execution. Hence we can define the rate $\mathrm{R}(S)$ of server S to be $\mathrm{R}(\mathrm{cli}(S))$. Since servers are themselves 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 Γ a set of servers, then $\mathrm{ser}(\mathrm{cli}(S)) = S$ and $\mathrm{cli}(\mathrm{ser}(\Gamma)) = \Gamma$.

As an example, consider Figure 3, where Γ 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 τ_1, and τ_2 and τ_3, respectively. If $\mathrm{R}(\tau_1) = 0.4$, $\mathrm{R}(\tau_2) = 0.2$ and $\mathrm{R}(\tau_3) = 0.1$, then $\mathrm{R}(S_1) = 0.4$ and $\mathrm{R}(S_2) = 0.3$. 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 $\mathrm{R}(S) = 0.7$.

We now define a *unit set* and *unit servers*, both of which can be feasibly scheduled on one processor.



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

Fig. 3. A two-server set. The notation $X^{(\mu)}$ means that $\mathrm{R}(X) = \mu$.

Definition III.2 (Unit Set/Unit Server)**.** *A set* Γ *of servers is a* unit set *if* $\mathrm{R}(\Gamma) = 1$. *The server* $\mathrm{ser}(\Gamma)$ *for a unit set* Γ *is a* unit server.

By Definition III.1, the execution requirement of a server S in any interval $[d_i, d_{i+1})$ equals $\mathrm{R}(S)(d_{i+1} - d_i)$, where d_i and d_{i+1} are consecutive deadlines in $\mathrm{D}(S)$. Then the workload for job J of server S with $J.r = d_i$ and $J.d = d_{i+1}$ equals $J.c = e(J, J.r) = \mathrm{R}(S)(J.d - J.r)$, just as with a "real" job. However, just as a server S is a proxy for its clients, so too are the "jobs" of S, which represent budget allocated to S so that its clients' jobs may execute. At each $d \in \mathrm{D}(S)$, when server S releases a job J_S, S replenishes this budget. At any given time t, that budget is just $e(J_S, t)$, where J_S is the current job of S.

It is not necessary for all arrival times to be known at the outset. We only require that there be no gaps or overlap between a task's jobs. Thus at any moment, we always know each task's next deadline. This is an important distinction; unlike periodic tasks where all deadlines are known at the outset, the fixed-rate task model allows for jobs whose deadlines are not known *a priori*.

Unlike previous approaches, tasks in RUN do not receive their proportional share between each system deadline. By reserving utilizations equal to the summed rates of their client tasks, servers enforce "weak proportional fairness." The execution of a server's job ensures that its set of clients collectively gets its proportional share of processor time between each server deadline, *i.e.,* between the deadlines of the clients of the server. This weak proportional fairness guarantees RUN's optimality and greatly reduces overhead compared to algorithms based on standard proportional fairness.

Even if a server meets all of its deadlines, it must use an appropriate scheduling policy to ensure that its clients meet theirs. We use EDF to ensure optimality within each server.

For example, consider two periodic tasks $\tau_1{:}(1/2, 2\mathbb{N}^*)$ and $\tau_2{:}(1/3, 3\mathbb{N}^*)$, with periods equal to 2 and 3 and utilizations $\mu(\tau_1) = 1/2$ and $\mu(\tau_2) = 1/3$, respectively. Assume start times of zero, as usual. Consider a server S scheduling these two tasks on a dedicated processor and let $\mathrm{D}(S) = \{2, 3, 4, 6, \ldots\}$. Thus, the budget of S during $[0, 2)$ equals $e(J_S, 0) = 2\,\mathrm{R}(S) = 5/3$; that is, S releases a virtual job J_S at time $t = 0$ with workload $5/3$ and deadline 2. Let Σ be a schedule of τ_1 and τ_2 in which S is meets this deadline. It follows that S acquires the processor for at least $5/3$ units of time during $[0, 2)$. Now, suppose that the scheduling policy used by S to schedule its client tasks gives higher priority to τ_2 at time 0. Then τ_2 will consume one unit of time before τ_1 begins its execution. Therefore, the remaining budget $e(J_S, 1) = 2/3$ will be insufficient to complete τ_1 by its deadline at 2. Thus a server can meet its

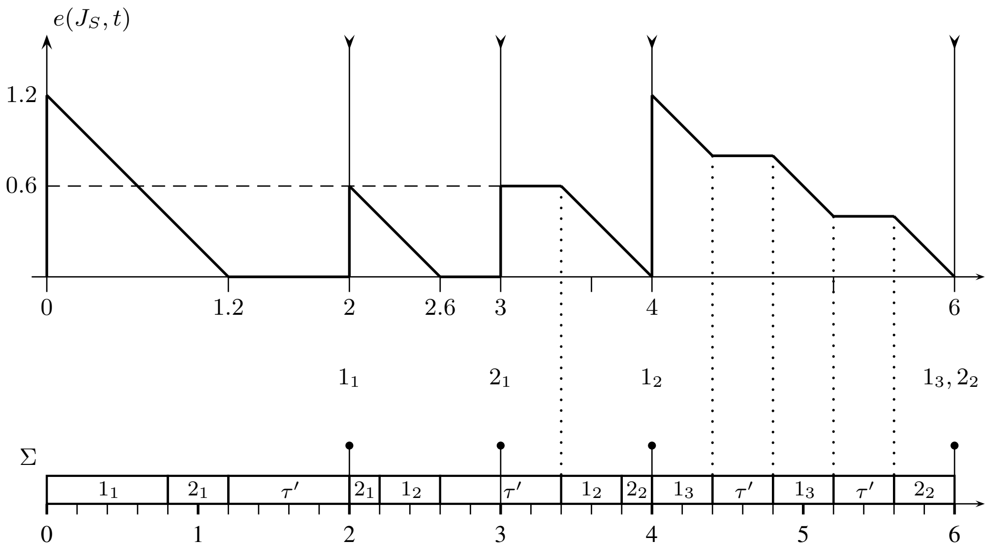


Fig. 4. Budget management and schedule of S; $\mathrm{cli}(S) = \{\tau_1{:}(0.4, 2\mathbb{N}^*), \tau_2{:}(0.2, 3\mathbb{N}^*)\}$ and $\mathrm{R}(S) = 0.6$; τ' represents execution of external events.

deadlines even when its clients do not.

B. EDF Server

Rule III.1 (EDF Server). *An* EDF server *is a server that schedules its client jobs with EDF.*

Consider a set of two periodic tasks $\mathcal{T} = \{\tau_1{:}(0.4, 2\mathbb{N}^*),\ \tau_2{:}(0.2, 3\mathbb{N}^*)\}$. Since $\mathrm{R}(\mathcal{T}) = 0.6 \leqslant 1$, we can define an EDF server S to schedule $\mathcal{T}$ such that $\mathrm{cli}(S) = \mathcal{T}$ and $\mathrm{R}(S) = 0.6$. Figure 4 shows both the evolution of $e(J_S, t)$ during interval $[0, 6)$ and the schedule Σ of $\mathcal{T}$ by S on a single processor. In this figure, i_j represents the j-th job of τ_i. During intervals $[1.2, 2)$, $[2.6, 3.4)$, $[4.4, 4.8)$ and $[5.2, 5.6)$, the execution of S is replaced with execution of external events represented by τ'.

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

Proof: By treating the servers in Γ 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.

Let $\mathrm{R}(\Gamma) = 1$ and $\eta_\Gamma(t, t')$ 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'. This quantity is bounded above by

$$\eta_\Gamma(t, t') \leqslant (t' - t) \sum_{S_i \in \Gamma} \mathrm{R}(S_i) = t' - t$$

It is known that there is no valid schedule for Γ if and only if there is some interval $[t, t')$ such that $\eta_\Gamma(t, t') > t' - t$ [8], [9]. Since this cannot happen, some valid schedule for Γ must exist. Because S schedules Γ using EDF and EDF is optimal [8], [10], S must produce a valid schedule.

Now, suppose that $\mathrm{R}(\Gamma) < 1$. We introduce a slack-filling task τ' where $D(\tau') = D(S)$ and $R(\tau') = 1 - R(S)$. We let $\Gamma' = \Gamma \cup \{\tau'\}$, and let S' be an EDF server for Γ'. Since $R(\Gamma') = 1$, S' produces a valid schedule for Γ'. Now consider the time interval $I_J = [J.r, J.d]$ for a job J of S. Since $D(\tau') = D(S)$, τ' also has a job J' where $J'.r = J.r$ and $J'.d = J.d$. Since S' produces a valid schedule, τ' and S do exactly $R(\tau')(J.d - J.r)$ and $R(S)(J.d - J.r)$ units of work, respectively, during I_J. Since there are no deadlines or releases between $J.r$ and $J.d$, the workload of τ' may be arbitrarily rearranged or subdivided within the interval I_J without compromising the validity of the schedule. We may do this so as to reproduce *any* scheduling of S where it meets its deadlines. Further, since S and S' both schedule tasks in Γ with EDF, S will produce the same *valid* schedule for Γ 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.

IV. Virtual Scheduling and RUN

We now describe the operations, DUAL and PACK, which iteratively reduce the number of processors in a multiprocessor system until a set of uniprocessor systems is derived. The schedules for these uniprocessor systems are produced on-line by EDF and from these the corresponding schedule for the original multiprocessor system is constructed.

The DUAL operation, detailed in Section IV-A, transforms a task τ into the dual task τ^*, whose execution time represents the idle time of τ. Since $\mathrm{R}(\tau^*) = 1 - \mathrm{R}(\tau)$, the DUAL operation reduces the total rate and the number of required processors in systems where most tasks have high rates.

Such high rate tasks are generated via the PACK operation, presented in Section IV-B. Sets of tasks whose rates sum to no more than one can be packed into servers, reducing the number of tasks and producing the high-rate tasks needed by the DUAL operation. Given this synergy, we compose the two operations into a single REDUCE operation, which will be defined in Section IV-C. As will be seen in Section IV-D, after a sequence of REDUCE operations, the schedule of the multiprocessor system can be deduced from the (virtual) schedules of the derived uniprocessor systems. The reduction from the original system to the virtual ones is carried out off-line. The generation of these various systems' schedules can be done on-line.

A. DUAL *Operation*

Our example of Figure 2 demonstrated dual tasks and schedules and enabled $m + 1$ tasks with total rate m to be scheduled on a uniprocessor dual system (as previously discussed in [11]). We now generalize these ideas in terms of servers.

Definition IV.1 (Dual Server). *The* dual server S^* *of a server* S *is a server with the same deadlines as* S *where* $\mathrm{R}(S^*)$ *equals* $1-\mathrm{R}(S)$. *If* Γ *is a set of servers, then its dual set* Γ^* *is the set of dual servers to those in* Γ, i.e., $S \in \Gamma$ *if and only if* $S^* \in \Gamma^*$.

The dual of a unit server, which has rate $\mathrm{R}(S) = 1$ and must execute continuously in order to meet its clients' deadlines, is a *null server*, which has rate $\mathrm{R}(S)=0$ and never executes.

Definition IV.2 (Dual Schedule). *Let* Γ *be a set of servers and* Γ^* *be its dual set. Two schedules* Σ *of* Γ *and* Σ^* *of* Γ^* *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.*

S, Γ, and Σ are referred to as *primal* relative to their duals S^*, Γ^*, and Σ^*. Note that $(\tau^*)^* = \tau$ and $(\Sigma^*)^* = \Sigma$, as expected with duality. We next observe that a schedule is valid precisely when its dual is valid.

Theorem IV.1 (Dual Validity). *Let* Γ *be a set of* $n = m + k$ *servers on* m *processors, scheduled by* Σ*, with* $k \geqslant 1$ *and* $\mathrm{R}(\Gamma) = m$*. Let* Γ^* *and* Σ^* *be their duals. Then* $\mathrm{R}(\Gamma^*) = k$*, and so* Γ^* *is feasible on* k *processors. Further,* Σ *is valid if and only if* Σ^* *is valid.*

Proof: First,

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

so k processors are sufficient to schedule Γ^*. Next, assume Σ is valid for Γ. We confirm that Σ^* is valid for Γ^* via Definitions II.3 and II.4.

Because Σ is a valid schedule on m processors and we assume full utilization, Σ always executes m distinct tasks. The remaining $k = n - m$ tasks are idle in Σ, and so are exactly the tasks executing in Σ^*. Hence Σ^* is always executing exactly k distinct tasks on its k (dual) processors. Since Σ is valid, any job J of server $S \in \Gamma$ does exactly $J.c = \mathrm{R}(S)(J.d - J.r)$ 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

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

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 Σ, J^* executes for $J^*.c$ in Σ^*. Consequently, J^* satisfies condition (ii) of Definition II.3 and also meets its deadline. Since this holds for all jobs of all dual servers, Σ^* is a valid schedule for Γ^*.

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

Once again, see Figure 2 for a simple illustration. We now summarize this dual scheduling rule for future reference.

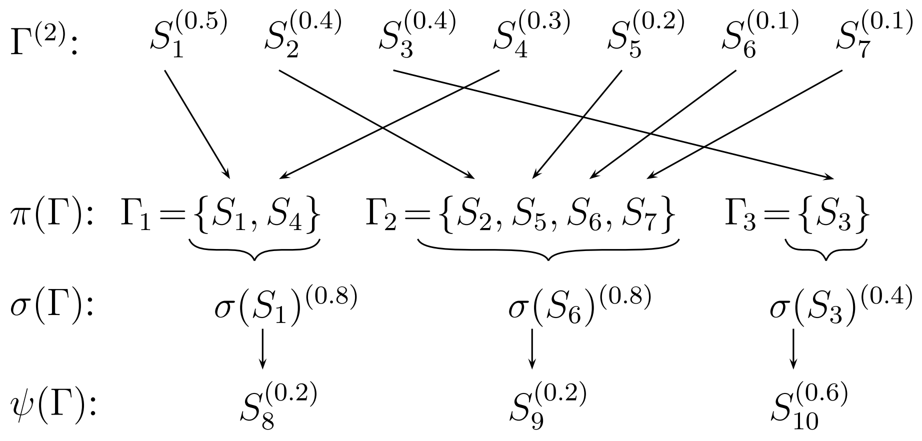

$\Gamma^{(2)}$:
$S_1^{(0.5)}$ $S_2^{(0.4)}$ $S_3^{(0.4)}$ $S_4^{(0.3)}$ $S_5^{(0.2)}$ $S_6^{(0.1)}$ $S_7^{(0.1)}$
$\pi(\Gamma)$: $\Gamma_1 = \{S_1, S_4\}$ $\Gamma_2 = \{S_2, S_5, S_6, S_7\}$ $\Gamma_3 = \{S_3\}$
$\sigma(\Gamma)$: $\sigma(S_1)^{(0.8)}$ $\sigma(S_6)^{(0.8)}$ $\sigma(S_3)^{(0.4)}$
$\psi(\Gamma)$: $S_8^{(0.2)}$ $S_9^{(0.2)}$ $S_{10}^{(0.6)}$

Fig. 5. Packing, PACK operation, and duality 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 $S_8 = \varphi \circ \sigma(S_1)$, $S_9 = \varphi \circ \sigma(S_6)$, $S_{10} = \varphi \circ \sigma(S_3)$. The notation $X^{(\mu)}$ means that $\mathrm{R}(X) = \mu$.

Rule IV.1 (Dual Server)**.** *At any time, execute in Σ the servers of Γ whose dual servers are not executing in Σ^*.*

Finally, we define the DUALoperation φ from a set of servers Γ to its dual set Γ^* as the bijection which associates a server S with its dual server S^*, *i.e.,* $\varphi(S) = S^*$. We adopt the convention of applying a function to subsets as well as elements. That is, if $f : A \to B$ and $A_1 \subseteq A$, we understand $f(A_1)$ to mean $\{f(a), a \in A_1\}$. For example, $\varphi(\Gamma) = \Gamma^*$.

Theorem IV.1 states that finding a valid schedule for a server set on m processors is equivalent to finding a valid schedule for its dual on $n-m$ virtual processors, which is advantageous whenever $n-m < m$. The PACK operation ensures that this is always the case.

B. PACK *Operation*

The DUAL operation reduces the number of processors whenever $n-m < m$. When $n - m \geqslant m$, the number of servers can be reduced by aggregating them into fewer servers with the PACK operation.

Definition IV.3 (Packing)**.** *Let Γ be a set of servers. A* packing *of Γ, denoted $\pi(\Gamma)$, is a partition of Γ into a collection of subsets $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_k\}$ such that $\mathrm{R}(\Gamma_i) \leqslant 1$ for all i and $\mathrm{R}(\Gamma_i) + \mathrm{R}(\Gamma_j) > 1$ for all $i \neq j$.*

An example of packing a set Γ of 7 servers into three sets Γ_1, Γ_2 and Γ_3, is illustrated by rows 1 and 2 of Figure 5.

Definition IV.4 (PACK operation)**.** *Let Γ be a set of servers and π a packing of Γ. We associate a server $\mathrm{ser}(\Gamma_i)$ with each subset $\Gamma_i \in \pi(\Gamma)$. The* PACK operation *σ is a function from Γ to these servers of $\pi(\Gamma)$ such that, if $\Gamma_i \in \pi(\Gamma)$ and $S \in \Gamma_i$, then $\sigma(S) = \mathrm{ser}(\Gamma_i)$. That is, $\sigma(S)$ is the aggregated server responsible for scheduling S.*

Rows 2 and 3 of Figure 5 show that $\sigma(S_1) = \mathrm{ser}(\Gamma_1)$, $\sigma(S_6) = \mathrm{ser}(\Gamma_2)$ and

$\sigma(S_3) = \mathrm{ser}(\Gamma_3)$. Also, if S_i and S_j are packed in the same subset Γ_k by π, then $\sigma(S_i) = \sigma(S_j) = \sigma(\Gamma_k)$. Hence, $\sigma(\Gamma) = \{\sigma(\Gamma_k), \Gamma_k \in \pi(\Gamma)\}$.

Definition IV.5 (Packed Server Set)**.** *A set of servers* Γ *is* packed *if it is a singleton, or if* $|\Gamma| \geqslant 2$ *and for any two distinct servers* S *and* S' *in* Γ*,* $\mathrm{R}(S) + \mathrm{R}(S') > 1$.

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

C. REDUCE *Operation*

We now compose the DUAL and PACK operations into the REDUCE operation. A sequence of reductions transforms a multiprocessor scheduling problem to a collection of uniprocessor scheduling problems. This transformation is the first half of the RUN algorithm.

Lemma IV.1. *If* Γ *is a packed set of servers and* σ *is the* PACK *operation associated with a packing* π *of* Γ*'s dual set* $\varphi(\Gamma)$*, then* $|\sigma\circ\varphi(\Gamma)| \leqslant \left\lceil \frac{|\Gamma|+1}{2} \right\rceil$.

Proof: Let $n = |\Gamma|$. Since Γ is packed, there is at most one server S in Γ such that $\mathrm{R}(S) \leqslant 1/2$. This implies that at least $n-1$ servers in $\varphi(\Gamma)$ have rates less than $1/2$. When these $n-1$ dual servers are packed, they will be, at a minimum, paired off. Thus, π will pack $\varphi(\Gamma)$ into at most $\lceil (n-1)/2 \rceil + 1$ subsets. Hence, $|\sigma\circ\varphi(\Gamma)| \leqslant \lceil (n+1)/2 \rceil$. ■

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 IV.6. *Given a set of servers* Γ *and a packing* π *of* Γ*, a* REDUCE operation *on a server* S *in* Γ*, denoted* $\psi(S)$*, is the composition of the* DUAL *operation* φ *with the* PACK *operation* σ *for* π*,* i.e., $\psi(S) = \varphi\circ\sigma(S)$.

Figure 5 illustrates the steps of the REDUCE operation ψ. As we intend to apply REDUCE repeatedly until we are left with only unit servers, we now define a *reduction sequence*.

Definition IV.7 (Reduction Level/Sequence)**.** *Let* $i \geqslant 1$ *be an integer,* Γ *a set of servers, and* S *a server in* Γ*. The operator* ψ^i *is recursively defined by* $\psi^0(S) = S$ *and* $\psi^i(S) = \psi\circ\psi^{i-1}(S)$*.* $\{\psi^i\}_i$ *is a* reduction sequence*, and the server system* $\psi^i(\Gamma)$ *is said to be at* reduction level i.

Theorem IV.2 proves that a reduction sequence on a server set Γ with $\mathrm{R}(\Gamma) = m$ always arrives at a collection of unit servers. Table II shows 10 servers transformed into a unit server via two REDUCE operations and a final PACK. Two unit servers appear before the terminal level (indicated in the table by $\mathbf{1}\!\rightarrow$). The dual of a unit server is a null server, which is packed into another server in the next step. This is unnecessary; a unit server can be assigned to a subset of processors and executed independently from all other tasks. We call the tasks in a unit server a *proper subset*, and, along with their assigned processors, a *proper subsystem.* Blank columns in Table II separate the

TABLE II
SAMPLE REDUCTION AND PROPER SUBSETS

	Server Rate									
$\psi^0(\Gamma)$	.6	.6	.6	.6	.6	.8	.6	.6	.5	.5
$\sigma(\psi^0(\Gamma))$	.6	.6	.6	.6	.6	.8	.6	.6	**1**$\rightarrow$	
$\psi^1(\Gamma)$	.4	.4	.4	.4	.4	.2	.4	.4	0	
$\sigma(\psi^1(\Gamma))$	.8		.8		.4	**1**$\rightarrow$				
$\psi^2(\Gamma)$	.2		.2		.6	0				
$\sigma(\psi^2(\Gamma))$	**1**									

TABLE III
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	

three proper subsystems. Separating proper subsystems yields more efficient scheduling because tasks in one subsystem do not impose events on or migrate to other subsystems. However, in this section we will just DUAL our unit servers and PACK these into other servers to simplify our proofs.

Lemma IV.2. *Let Γ be a packed set of servers, not all of which are unit servers. If $\mathrm{R}(\Gamma)$ is a positive integer, then $|\Gamma| \geqslant 3$.*

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

Theorem IV.2 (Reduction Convergence)**.** *Let Γ be a set of servers where $\mathrm{R}(\Gamma)$ is a positive integer. Then for some $p \geqslant 0$, $\sigma(\psi^p(\Gamma))$ is a set of unit servers.*

Proof: Let $\Gamma^k = \psi^k(\Gamma)$ and $\Gamma^k_\sigma = \sigma(\Gamma^k)$, and suppose that $\mathrm{R}(\Gamma^k_\sigma)$ is a positive integer. If Γ^k_σ is a set of unit servers, then $p = k$ and we're done. Otherwise, according to Lemma IV.2, $|\Gamma^k_\sigma| \geqslant 3$. Since Γ^k_σ is a packed set of servers, Lemma IV.1 tells us that $|\Gamma^{k+1}_\sigma| = |\sigma \circ \varphi(\Gamma^k_\sigma)| \leqslant \left\lceil \frac{|\Gamma^k_\sigma|+1}{2} \right\rceil$.

Since $\lceil (x+1)/2 \rceil < x$ for $x \geqslant 3$ and $|\Gamma^k_\sigma| \geqslant 3$, we have that $|\Gamma^{k+1}_\sigma| < |\Gamma^k_\sigma|$. Also, since $\mathrm{R}(\Gamma^k_\sigma)$ is a positive integer (and less than $|\Gamma^k_\sigma|$, since Γ^k_σ are not all unit servers), Theorem IV.1 tells us that $\mathrm{R}(\varphi(\Gamma^k_\sigma))$ is also a positive integer; as is $\mathrm{R}(\Gamma^{k+1}_\sigma)$, since packing does not change total rate. Thus Γ^{k+1}_σ satisfies the same conditions as Γ^k_σ, but contains fewer servers.

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 IV.2 tells us that Γ^k_σ must be a set of unit servers; $p = k$. ■

In other words, a reduction sequence on any set of servers eventually produces a set of unit servers. We show how to schedule a unit server's proper subsystem in the next section. However, ψ is not a well-defined function; it is a mapping whose outcome is dependent on the packings used. Table III 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 IV.2 implicitly proves that all packings "work"; they all lead to a correct reduction to *some* set of unit servers.

D. Scheduling and RUN

Now that we have transformed a multiprocessor system into one or more uniprocessor systems, we show how to schedule them. The basic idea is to use the dual schedules to find primal schedules and use EDF servers to schedule client servers and tasks. Theorem IV.2 says that a reduction sequence produces a collection of one or more unit servers. As shown in Table II, the original task set may be partitioned into the proper subsets represented by these unit servers, which may be scheduled independently. In this section, we assume that $\mathcal{T}$ is a proper subset, *i.e.,* that it is handled by a single unit server at the terminal reduction level.

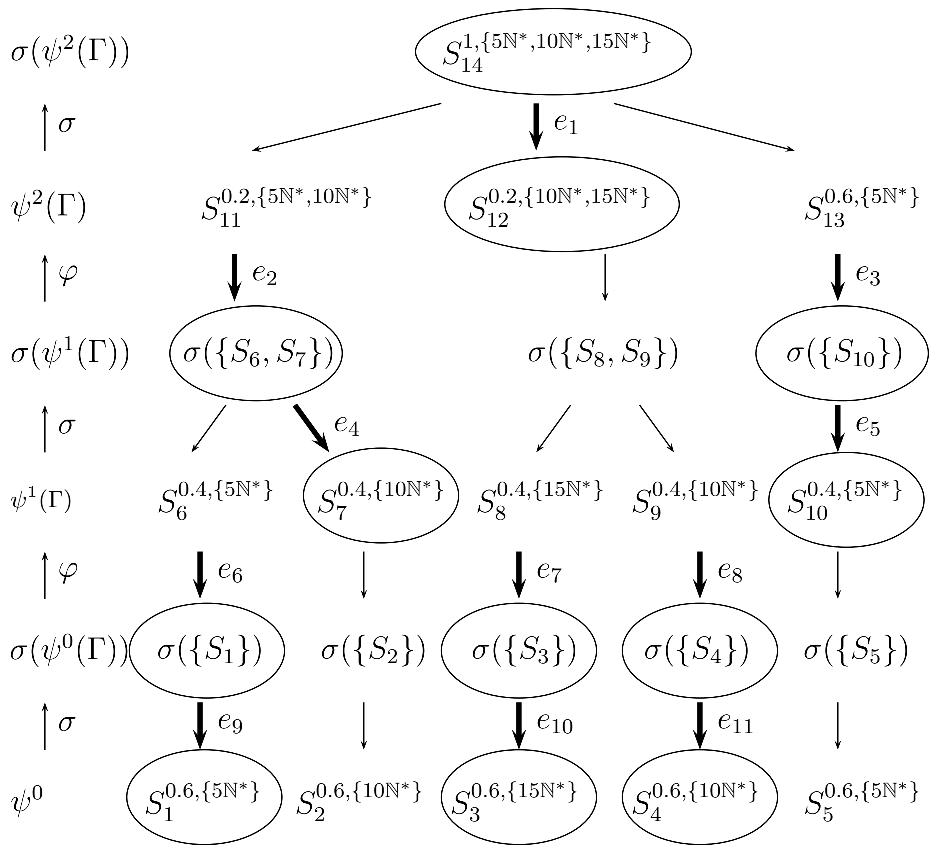
The scheduling process is illustrated by inverting the reduction tables from the previous section and creating a *server tree* (see Figure 6). The unit server is the root, which represents the top level virtual uniprocessor system. The root's children are the unit server's clients, which are scheduled by EDF. In Figure 6, the servers executing at each level at time $t = 4$ are circled. The schedule for $\mathcal{T}$ (the leaves of the tree) is obtained by propagating the schedule down the tree using Rules III.1 (schedule clients with EDF) and IV.1 (use Σ^* to find Σ). In terms of the server tree, these rules may be restated as:

Rule IV.2 (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 IV.3 (Dual Server)**.** *Execute (circle) the child (packed server) of a dual server if and only if the dual server is* not *executing (not circled).*

Figure 6 shows this procedure on the first proper subset found in Table II To the five tasks with rate 0.6, we assign the deadline sets $5\mathbb{N}^*, 10\mathbb{N}^*, 15\mathbb{N}^*, 10\mathbb{N}^*$, and $5\mathbb{N}^*$, respectively. Rule IV.2 is seen in the tree edges $\{e_1, e_4, e_5, e_9, e_{10}, e_{11}\}$. Rule IV.3 is seen in the tree edges $\{e_2, e_3, e_6, e_7, e_8\}$. With these two simple rules, at any time t, we can determine which tasks in $\mathcal{T}$ should be executing by circling the root and propagating circles down the tree into the leaves. In practice, we only need to execute the rules when some subsystem's EDF scheduler generates a scheduling event (*i.e.,* WORK COMPLETE or JOB RELEASE). Figure 6 shows the scheduling decision process at $t = 4$, and Figure 7 shows the full schedule for all three reduction levels for ten time units.

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



$\sigma(\psi^2(\Gamma))$
σ
$S_{14}^{1,\{5\mathbb{N}^*,10\mathbb{N}^*,15\mathbb{N}^*\}}$
e_1
$\psi^2(\Gamma)$
$S_{11}^{0.2,\{5\mathbb{N}^*,10\mathbb{N}^*\}}$
$S_{12}^{0.2,\{10\mathbb{N}^*,15\mathbb{N}^*\}}$
$S_{13}^{0.6,\{5\mathbb{N}^*\}}$
φ
e_2
e_3
$\sigma(\psi^1(\Gamma))$
$\sigma(\{S_6,S_7\})$
$\sigma(\{S_8,S_9\})$
$\sigma(\{S_{10}\})$
e_4
e_5
$\psi^1(\Gamma)$
$S_6^{0.4,\{5\mathbb{N}^*\}}$
$S_7^{0.4,\{10\mathbb{N}^*\}}$
$S_8^{0.4,\{15\mathbb{N}^*\}}$
$S_9^{0.4,\{10\mathbb{N}^*\}}$
$S_{10}^{0.4,\{5\mathbb{N}^*\}}$
e_6
e_7
e_8
$\sigma(\psi^0(\Gamma))$
$\sigma(\{S_1\})$
$\sigma(\{S_2\})$
$\sigma(\{S_3\})$
$\sigma(\{S_4\})$
$\sigma(\{S_5\})$
e_9
e_{10}
e_{11}
ψ^0
$S_1^{0.6,\{5\mathbb{N}^*\}}$
$S_2^{0.6,\{10\mathbb{N}^*\}}$
$S_3^{0.6,\{15\mathbb{N}^*\}}$
$S_4^{0.6,\{10\mathbb{N}^*\}}$
$S_5^{0.6,\{5\mathbb{N}^*\}}$

Fig. 6. Reduction tree used to schedule $\mathcal{T} = \{S_1, \ldots, S_5\}$ from Table II by Rules IV.2 and IV.3 at scheduling instant 4. The notation $S_i^{\mu,D}$ means that $\mathrm{R}(S_i) = \mu$ and $\mathrm{D}(S_i) = D$.

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 Section III-A.

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

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;
```

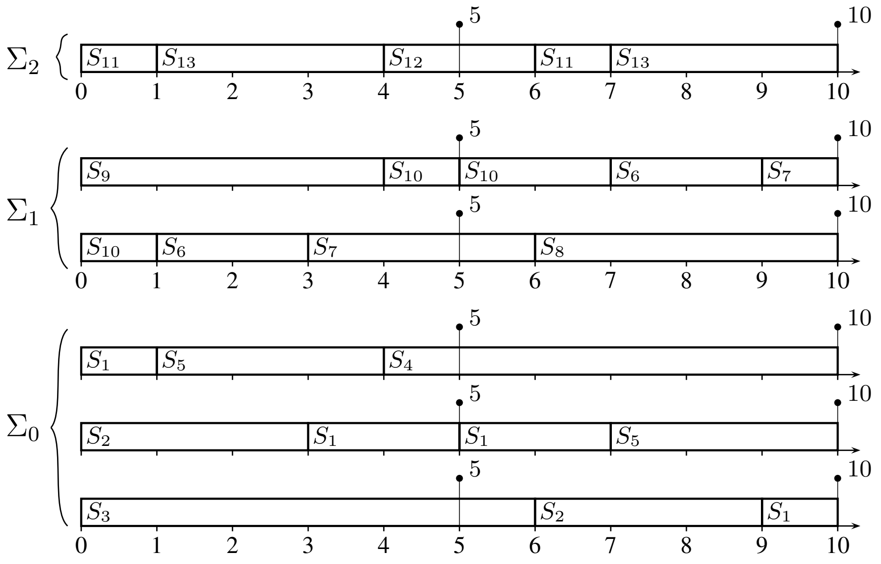


Fig. 7. $\mathcal{T} = \{S_1, S_2, S_3, S_4, S_5\}$ with $S_1 = \mathrm{ser}(3/5, 5\mathbb{N}^*)$, $S_2 = \mathrm{ser}(3/5, 10\mathbb{N}^*)$, $S_3 = \mathrm{ser}(3/5, 15\mathbb{N}^*)$, $S_4 = \mathrm{ser}(3/5, 10\mathbb{N}^*)$, $S_5 = \mathrm{ser}(3/5, 5\mathbb{N}^*)$. Σ_0 is the schedule of $\mathcal{T}$ on 3 physical processors. Σ_1 is the schedule of $\psi(\mathcal{T}) = \{S_6, S_7, S_8, S_9, S_{10}\}$ on 2 virtual processors, and Σ_2 is the schedule of $\psi^2(\mathcal{T}) = \{S_{11}, S_{12}, S_{13}\}$ on 1 virtual processor.

to as the RUN algorithm and is summarized in Algorithm 1. We now finish proving it correct.

Theorem IV.3 (Reduction Schedule)**.** *If Γ is a proper set under the reduction sequence $\{\psi^i\}_{i\leqslant p}$, then the RUN algorithm produces a valid schedule Σ for Γ.*

Proof: Again, let $\Gamma^k = \psi^k(\Gamma)$ and $\Gamma^k_\sigma = \sigma(\Gamma^k)$ with $k < p$. Also, let Σ^k and Σ^k_σ be the schedules generated by RUN for Γ^k and Γ^k_σ, respectively. Finally, let $\mu^k = \mathrm{R}(\Gamma^k) = \mathrm{R}(\Gamma^k_\sigma)$, which, as seen in the proof of Theorem IV.2, is always an integer.

We will work inductively to show that schedule validity propagates down the reduction tree, *i.e.,* that the validity of Σ^{k+1} implies the validity of Σ^k. Suppose that Σ^{k+1} is a valid schedule for $\Gamma^{k+1} = \varphi(\Gamma^k_\sigma)$ on μ^{k+1} processors, where $k+1 \leqslant p$. Since $k < p$, Γ^k_σ is not the terminal level set, and so must contain more than one server, as does its equal-sized dual Γ^{k+1}. Further, since Γ is a proper set under our reduction, none of these servers can be unit servers and so $|\Gamma^{k+1}| > \mu^{k+1}$. The conditions of Theorem IV.1 are satisfied (where $n = |\Gamma^{k+1}|$, $m = \mu^{k+1}$, and $k > 1$), so it follows from our assumption that Σ^{k+1} is valid that $\Sigma^k_\sigma = (\Sigma^{k+1})^*$ is a valid schedule for Γ^k_σ on μ^k processors. Also, since Γ^k_σ is a collection of aggregated servers for Γ^k, it follows from Theorem III.1 that Σ^k is a valid schedule for Γ^k (*i.e.,* scheduling the servers in Γ^k_σ correctly ensures that all of their client tasks in Γ^k are also scheduled correctly). Thus the validity of Σ^{k+1} implies the validity of Σ^k, as desired.

Since uniprocessor EDF generates a valid schedule Σ^p for the clients of the unit server at terminal reduction level p, it follows inductively that $\Sigma = \Sigma^0$ is valid for Γ on $\mathrm{R}(\Gamma)$ processors. ■

V. Assessment

A. Implementation Details

Our implementation of RUN uses the worst-fit bin-packing heuristic, which runs in $O(n \log n)$ time. Our reduction procedure partitions off proper subsystems as unit servers are found. At each scheduler invocation, once the set of m running tasks is determined (as in Figure 6), we use a simple greedy task-to-processor assignment scheme. In three passes through these m tasks, we (i) leave executing tasks on their current processors, (ii) assign idle tasks to their last-used processor, when available, to avoid unnecessary migrations, and (iii) assign remaining tasks to free processors arbitrarily.

Duality is only defined for task sets with 100% utilization. Dummy tasks fill in the difference when needed. 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 $m - \mathrm{R}(\mathcal{T})$) 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 Figure 6 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 $\mathrm{R}(S_1^d) = 0.4$ (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. Since S_1 and S_2 never need preempt or migrate, 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.

Worst-fit 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. Worst-fit bin packing can be replaced with any other partitioning scheme that (i) uses additional "bins" when a proper partitioning onto m processors is not found, and (ii) creates a packed server set. And any optimal uniprocessor scheduling algorithm can be substituted for EDF. In this way, the RUN scheme can be used as an extension of different partitioned scheduling algorithms, but one that could, in theory, handle cases when a proper partition on m processors can't be found.

B. 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.

Lemma V.1. *If Γ is a packed set of at least 2 servers, then $\mathrm{R}(\Gamma) > |\Gamma|/2$.*

Proof: Let $n = |\Gamma|$, and let $\mu_i = \mathrm{R}(S_i)$ for $S_i \in \Gamma$. Since Γ is packed, there exists at most one server in Γ, say S_n, such that $\mu_n \leqslant 1/2$; 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 $\mathrm{R}(\Gamma) = \sum_{i=1}^{n} \mu_i > n/2$. ■

Theorem V.1 (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 $\{\psi^i\}_{i\leqslant p}$ be a reduction sequence on $\mathcal{T}$, where p is the terminal level described in Theorem IV.2. Lemma IV.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 $T(n) \leqslant T(n/2) + O(f(n)+n)$, which gives $T(n) = O(f(n))$.

Since $\mathcal{T}$ is a full utilization task set, $\mathrm{R}(\mathcal{T}) = m$. If we let $n' = |\sigma(\mathcal{T})|$, Lemma V.1 tells us that $m = \mathrm{R}(\mathcal{T}) = \mathrm{R}(\sigma(\mathcal{T})) > n'/2$. But as $\sigma(\mathcal{T})$ is just the one initial packing, it follows that p also is $O(\log n')$, and hence $O(\log m)$. ■

Theorem V.2 (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's 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 $n + n + n/2 + n/4 + \cdots \approx 3n$.

Next, consider the scheduling process described by Rules IV.2 and IV.3. The comparison of clients performed by EDF in Rule IV.2 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 IV.3 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 $O(m) \leqslant O(n)$ time. Therefore, each scheduler invocation is accomplished in $O(n)$ time.

Since we only invoke the scheduler at WORK COMPLETE or JOB RELEASE events, 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 V.1). Thus j real jobs result in no more than $jO(\log m)$ virtual jobs, so a time interval where j jobs are released will see a total scheduling overhead of $O(jn \log m)$. ■

C. 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. Because all scheduling decisions are made by EDF, we need only consider the preemptions caused by two types of scheduling events: *work complete events* (W.C.E.), and *job release events* (J.R.E.) (which occur concurrently with job deadlines).

Lemma V.2. *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 V.3. *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. only causes a context switch when the arriving job J_i^*, say from task τ^*, has an earlier deadline than, and replaces, the previously running job. However, if J_i^* *starts* executing at time $J_i^*.r$ in the dual, then τ'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 its work is complete, 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 V.2, 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 V.4. *A context switch at any level of the server tree causes exactly one context switch between two original leaf tasks in $\mathcal{T}$.*

Proof: We proceed by induction, 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.*, $C_+ = S_+^*$

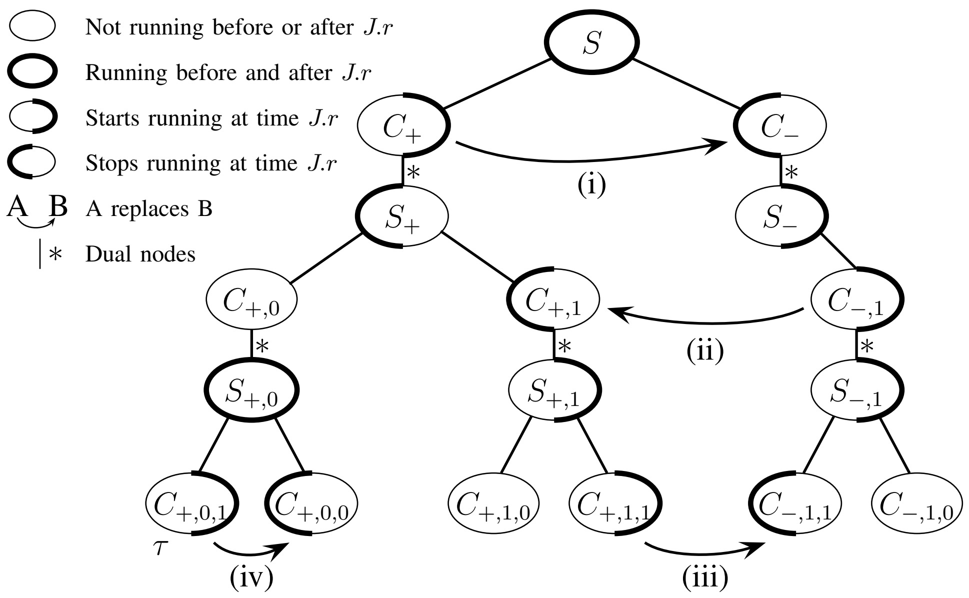


Fig. 8. **Two Preemptions from one JOB RELEASE**
In this 3-level server tree, a job release by τ corresponds to a job release and context switch at the top level (i), which propagates down to the right of the tree (ii, iii). That same job release by τ can cause it to preempt (iv) another client $C_{+,0,0}$ of its parent server $S_{+,0}$.

and $C_- = S_-^*$), so that S_- replaces S_+ (see Figure 8 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 $C_{-,1}$, gets executed. 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 Figure 8, (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 V.5. *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 τ releases job J at time $J.r$. This causes a job release at each ancestor server node above τ in the server tree (*i.e.,* on the path from leaf τ to the root). We will use Figure 8 for reference. Let S be the highest (furthest reduction level) ancestor server of τ 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 V.4, 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)). However, as no client of S_+ remains running at time $J.r$, the arrival of a job for τ'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 8 shows two context switches, (iii) and (iv), in $\mathcal{T}$ that result from a single J.R.E. of τ. One is caused by a job release by τ's ancestor child of the root, which propagates down to another part of the tree (iii). τ's parent server is not affected by this, stays running, and allows τ to preempt its sibling client when its new job arrives (iv).

While S is shown as the root and τ as a leaf in Figure 8, this argument would still apply if there were additional nodes above and below those shown, and τ were a descendant of node $C_{+,0,1}$. If there were additional levels, then τ'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 τ can cause at most $\lceil (p+1)/2 \rceil$ preemptions in $\mathcal{T}$. ■

Theorem V.3. *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 V.3. Otherwise, we use Lemma V.2 to count preemptions based on jobs from $\mathcal{T}$ and the two EDF event types. By Lemma V.5, 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 τ's p ancestors, and each of these p jobs may have a W.C.E. which causes (at most, by Lemma V.4) 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 $p = \log m$ by Theorem V.2. ■

In our simulations, we almost never observed a task set that required more than two reductions. For $p=2$, Theorem V.3 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: $\mathcal{T} = \{(.57, 4000), (.58, 4001), (.59, 4002), (.61, 4003), (.63, 4004), (.02, 3)\}$. 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.

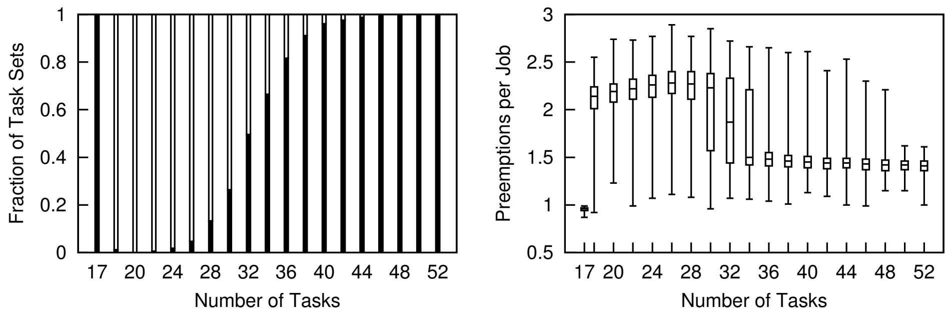


Fig. 9. Fraction of task sets requiring 1 (filled box) and 2 (empty box) reduction levels; Distributions of the average number of preemptions per job, their quartiles, and their minimum and maximum values. All RUN simulations on 16 processor systems at full utilization.

D. Simulation

We have evaluated RUN via extensive simulation using task sets generated for various levels of n tasks, m processors, and total utilization $\mathrm{R}(\mathcal{T})$. Task rates were generated in the range of $[0.01, 0.99]$ following the Emberson procedure [12] using the aleatory task generator [13]. 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 generated one thousand random n-task sets for each value $n = 17, 18, 20, 22, \ldots, 52$ (we actually took n up to 64, but results were nearly constant for $n \geqslant 52$). Each task set fully utilizes a system with 16 processors. We measured the number of reduction levels and the number of preemption points. Job completion is not considered a preemption point.

Figure 9(a) shows the number of reduction levels; none of the task sets generated require more than two reductions. For 17 tasks, only one level is necessary, as seen in Figure 2, and implied by Theorem IV.1. One or two levels are needed for $n \in [18, 48]$. None of our observed task sets require a second reduction for $n > 48$. 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.

The box-plot in Figure 9(b) shows the distribution of preemption points 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 $n \geqslant 36$, the median preemption count stays nearly constant just below 1.5. Even in the worst case, no task set ever incurs more than 2.8 preemptions per job on average.

Next, we ran comparison simulations against other optimal algorithms. In Figure 10, we count migrations and preemptions made by RUN, LLREF [5], EKG [6] and DP-Wrap [7] (with these last two employing the simple *mirroring*

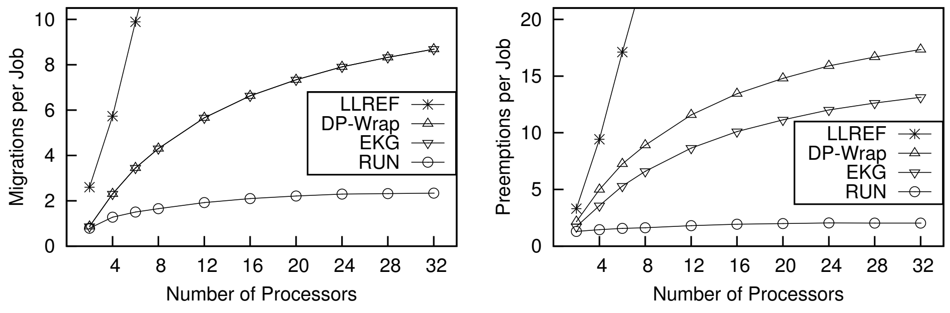


Fig. 10. Migrations- and preemptions-per-job by LLREF, DP-Wrap, EKG, and RUN as number of processors m varies from 2 to 32, with full utilization and $n = 2m$ tasks. Note: DP-Wrap and EKG have the same migration curves.

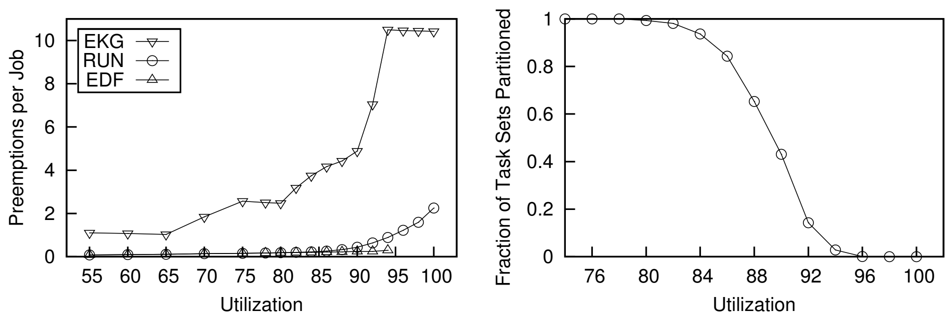


Fig. 11. Preemptions per job for EKG, RUN, and Partitioned EDF as utilization varies from 55 to 100%, with 24 tasks on 16 processors; Partitioning success rate for worst-fit bin packing under the same conditions.

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 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 Figure 11'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.

VI. 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 PPID model. If all tasks share the same deadline, the system can be optimally scheduled with very low implementation cost [2]. Optimality was first achieved without this restriction by *pfair* [3], 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 [4]–[7] 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 [5], [14] 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 [7], [15]. 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.

Other recent works have used the *semi-partitioning* approach to limit migrations [6], [16]–[19]. 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 uses a very weak version of proportional fairness: 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 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 [11] is generalized by our approach. The concept of task servers has been extensively used to provide a mechanism to schedule soft real-time tasks [20], 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 [21], [22]. Other server mechanisms have been designed for multiprocessor systems, *e.g.,* [6], [16], [23]. Unlike such approaches, the mechanism described here works as if each server were a uniprocessor system providing a useful scheduling framework which hides some complexities related to the multiprocessor scheduling problem.

VII. 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 worst-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.

Acknowledgment

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

References

[1] G. Koren, A. Amir, and E. Dar, “The power of migration in multi-processor scheduling of real-time systems,” in *ACM-SIAM symposium on Discrete algorithms*, ser. SODA ’98, 1998, pp. 226–235.

[2] R. McNaughton, “Scheduling with deadlines and loss functions,” *Management Science*, vol. 6, no. 1, pp. 1–12, 1959.

[3] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, “Proportionate progress: A notion of fairness in resource allocation,” *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[4] D. Zhu, D. Mossé, and R. Melhem, “Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary?” in *IEEE RTSS*, 2003, pp. 142–151.

[5] H. Cho, B. Ravindran, and E. D. Jensen, “An optimal real-time scheduling algorithm for multiprocessors,” in *IEEE RTSS*, 2006, pp. 101–110.

[6] B. Andersson and E. Tovar, “Multiprocessor Scheduling with Few Preemptions,” in *IEEE Embedded and Real-Time Computing Systems and Applications*, 2006, pp. 322–334.

[7] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, “DP-FAIR: a simple model for understanding optimal multiprocessor scheduling,” in *IEEE ECRTS*, 2010, pp. 3–13.

[8] S. Baruah and J. Goossens, “Scheduling real-time tasks: Algorithms and complexity,” in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, J. Y.-T. Leung, Ed. Chapman Hall/CRC Press, 2004.

[9] S. Baruah, A. Mok, and L. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor,” in *IEEE RTSS*, 1990, pp. 182 –190.

[10] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogram in a hard real-time environment,” *Journal of ACM*, vol. 20, no. 1, pp. 40–61, 1973.

[11] G. Levin, C. Sadowski, I. Pye, and S. Brandt, “SNS: a simple model for understanding optimal hard real-time multi-processor scheduling,” Univ. of California, Tech. Rep. UCSC-SOE-11-09, 2009.

[12] P. Emberson, R. Stafford, and R. I. Davis, “Techniques for the synthesis of multiprocessor tasksets,” in *WATERS*, 2010, pp. 6–11.

[13] ——, “A taskset generator for experiments with real-time task sets,” http://retis.sssup.it/waters2010/data/taskgen-0.1.tar.gz, Jan. 2011.

[14] K. Funaoka, S. Kato, and N. Yamasaki, “Work-conserving optimal real-time scheduling on multiprocessors,” in *IEEE ECRTS*, 2008, pp. 13–22.

[15] S. Funk, “An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines,” *Real-Time Syst.*, vol. 46, pp. 332–359, 2010.

[16] B. Andersson, K. Bletsas, and S. Baruah, “Scheduling arbitrary-deadline sporadic task systems on multiprocessors,” in *IEEE RTSS*, 2008, pp. 385–394.

[17] A. Easwaran, I. Shin, and I. Lee, “Optimal virtual cluster-based multiprocessor scheduling,” *Real-Time Syst.*, vol. 43, no. 1, pp. 25–59, 2009.

[18] S. Kato, N. Yamasaki, and Y. Ishikawa, “Semi-partitioned scheduling of sporadic task systems on multiprocessors,” in *IEEE ECRTS*, 2009, pp. 249–258.

[19] E. Massa and G. Lima, “A bandwidth reservation strategy for multiprocessor real-time scheduling,” in *IEEE RTAS*, 2010, pp. 175 –183.

[20] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, 2000.

[21] Z. Deng, J. W.-S. Liu, and J. Sun, “Scheme for Scheduling Hard Real-time Applications in Open System Environment,” in *ECRTS*, 1997, pp. 191–199.

[22] M. Spuri and G. Buttazzo, “Scheduling aperiodic tasks in dynamic priority systems,” *Real-Time Syst.*, vol. 10, no. 2, pp. 179–210, 1996.

[23] M. Moir and S. Ramamurthy, “Pfair scheduling of fixed and migrating periodic tasks on multiple resources,” in *IEEE RTSS*, 1999, pp. 294 –303.