Is Semi-Partitioned Scheduling Practical?*

Andrea Bastoni, Björn B. Brandenburg, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Semi-partitioned schedulers are—in theory—a particularly promising category of multiprocessor real-time scheduling algorithms. Unfortunately, issues pertaining to their implementation have not been investigated in detail, so their practical viability remains unclear. In this paper, the practical merit of three EDF-based semi-partitioned algorithms is assessed via an experimental comparison based on real-time schedulability under consideration of real, measured overheads. The presented results indicate that semi-partitioning is indeed a sound and practical idea. However, several problematic design choices are identified as well. These shortcomings and other implementation concerns are discussed in detail.

1 Introduction

The advent of multicore technologies has led to a surge of new research on multiprocessor real-time scheduling algorithms. When devising such algorithms and associated analysis, the goal is to enable *schedulable* systems to be produced. In a *hard real-time* (*HRT*) system, the term "schedulable" means that deadlines can never be missed. In contrast, in a *soft real-time* (*SRT*) system, occasional misses are tolerable, provided deadline tardiness is bounded.[1]

Most prior work on multiprocessor real-time scheduling has focused on *partitioned* and *global* approaches. Under partitioning, each task is statically assigned to a processor and per-processor schedulers are used. Under global scheduling, a system-wide run queue is used and task migration is allowed. Partitioned approaches incur lower runtime overheads (*e.g.*, no migrations and less run-queue contention), but require that a bin-packing-like problem be solved to assign tasks to processors; because of this, caps on total system utilization must be enforced to ensure that all tasks are schedulable (HRT or SRT). Such loss can be avoided by using global approaches, for both HRT [29] and SRT [26] systems. However, such approaches entail higher runtime costs.

Semi-partitioned scheduling was proposed by Anderson *et al.* [1] as a compromise between pure partitioned and global scheduling. Semi-partitioning extends partitioned scheduling by allowing a small number of tasks to migrate, thereby improving schedulability. Such tasks are called *migratory*, in contrast to *fixed* tasks that do not migrate. The original work on semi-partitioning [1, 2] was directed at SRT systems. Subsequently, other authors developed semi-partitioned algorithms for HRT systems [3, 4, 11, 23, 25]. The common goal in all of this work is to circumvent the algorithmic limitations and resulting capacity loss of partitioning while avoiding the overhead of global scheduling by limiting migrations.

At first glance, semi-partitioned algorithms seem rather challenging to implement, as they require separate per-processor run queues, but still require frequent migrations. The resulting cross-processor coordination could yield high scheduling costs. Worse, recent experimental evidence suggests that on some recent multicore platforms, (worst-case) preemption and migration costs do not differ substantially [8, 9], which calls into question the value of favoring preemptions over migrations.

The premise of semi-partitioned scheduling is fundamentally driven by practical concerns, yet its practical viability is virtually unexplored. Are complex semi-partitioned algorithms still preferable over straightforward partitioning when overheads are factored in? Do semi-partitioned schedulers actually incur significantly less overhead than global ones? In short, are the scheduling-theoretic gains of semi-partitioned scheduling worth the added implementation complexity?

Contributions. This paper presents the first in-depth study to address this issue of practicality by evaluating three semi-partitioned algorithms, EDF-fm, EDF-WM, and NPS-F (and its "clustered" variant C-NPS-F—see Sec. 3), under consideration of real, measured overheads. Our findings show that semi-partitioned scheduling is indeed a sound and practical approach for both HRT and SRT systems (Sec. 4). However, we also identify several shortcomings in the evaluated algorithms, in particular with regard to when and how migrations occur, and how tasks are assigned to processors. Based on these observations, we distill several design principles to aid in the future development of practical schedulers (Sec. 4.5).

Before describing our experimental setup and presenting our findings in detail, we first discuss relevant background.

2 Background

We focus herein on the scheduling of a system τ of *sporadic tasks*, $T_1, \ldots, T_n$, on m identical processors $P_1, \ldots, P_m$. Each task T_i is specified by its *worst-case execution time* e_i, its *period* p_i, and its (relative) *deadline* d_i. The j^{th} job (instance) of task T_i is denoted T_i^j. Such a job T_i^j becomes available for execution at its *release time* r_i^j and should complete by its

*Work supported by AT&T and IBM Corps.; NSF grants CNS 0834270 and CNS 0834132; ARO grant W911NF-09-1-0535; and AFOSR grant FA 9550-09-1-0549.

[1] Other notions of SRT correctness exist but are not considered herein.

deadline $r_i^j + d_i$; otherwise it is *tardy*. The spacing between r_i^j and r_i^{j+1} must satisfy $r_i^{j+1} \geq r_i^j + p_i$. A tardy job T_i^j does not alter r_i^{j+1}, but T_i^{j+1} cannot execute until T_i^j completes. For conciseness, we use $T_i = (e_i, p_i)$ to denote the parameters of a task. A task T_i is called an *implicit deadline* (resp., *constrained deadline*) task if $d_i = p_i$ (resp., $d_i \leq p_i$). If neither of these conditions applies, then T_i is called an *arbitrary deadline* task.

Task T_i's *utilization*, $u_i = e_i/p_i$, reflects the total processor share that it requires; the *total utilization* of the system is given by $U = \sum_{i=1}^{n} u_i$. In the semi-partitioned algorithms we consider, a task's utilization may be split among multiple processors. We let $s_{i,j}$ denote the fraction (or *share*) that a task T_i requires on processor P_j, where $\sum_{1 \leq j \leq m} s_{i,j} = u_i$.

Letting τ_j be the set of tasks assigned to processor P_j, the *assigned capacity* on P_j is $c_j = \sum_{T_i \in \tau_j} s_{i,j}$. The *available capacity* on P_j is thus $1 - c_j$.

2.1 Related Work

EDF-fm, EDF-WM, and NPS-F (and its clustered variant, C-NPS-F) are described in Sec. 3. Other EDF-based semi-partitioned algorithms that have been proposed include precursors to NPS-F [3, 4, 5, 10] and to EDF-WM [22, 23] and algorithms that share commonalities with these precursors [27].

To our knowledge, *detailed* runtime overheads affecting semi-partitioned algorithms have never been measured before within a real operating system (OS) and the impact of such overheads on the schedulability of these algorithms has never been assessed. Nonetheless, some simulation-based studies (without consideration of overheads) have been done. In [25], EDF-SS [4], which is a precursor of NPS-F, is compared to EDF-WM: EDF-SS exhibited less schedulability-related capacity loss than EDF-WM in the majority of the tested cases, at the cost of many more context switches. Given the lower number of context switches, we believe that EDF-WM is a better candidate than EDF-SS for an implementation-oriented study. EDF-WMR, which is a variant of EDF-WM that supports *reservations*, has been implemented within the AIRS framework [21], but detailed scheduling overheads or schedulability results were not reported. A proof-of-concept implementation of an algorithm called EKG-sporadic [3], which is also a precursor of NPS-F, has been discussed in a technical report [6].

Fixed-priority semi-partitioned algorithms have also been proposed (*e.g.* [19, 24]). From a schedulability perspective, such algorithms are generally inferior to EDF-based ones, so we defer their evaluation to future work.

2.2 Operating System and Hardware Capabilities

To better understand the design of the investigated algorithms, some knowledge of the services provided by the OS and by the hardware platform is required.

We implemented the algorithms evaluated in this study within LITMUS$^{\mathrm{RT}}$ [30], a real-time extension of the Linux kernel that allows schedulers to be developed as plugin components. We used LITMUS$^{\mathrm{RT}}$ 2010.2, which is based on Linux 2.6.34. The evaluated plugins are publicly available at [30].

Run queues. Many modern operating systems (including Linux) provide a scheduling framework that employs per-processor run queues. Holding a run-queue lock gives the owner the right to modify not only the run-queue state, but also the state of all tasks in the run queue.

Migrating a task under this locking rule requires local and remote run-queue locks to be acquired. Under scheduling algorithms that allow *concurrent* migrations, complex coordination is required to ensure that deadlocks do not occur and that migratory tasks will be executed by a single processor only (*i.e.*, only one processor may use a task's process stack). Algorithms where the likelihood of simultaneous scheduling decisions is high may thus entail rather high scheduling overheads.

Inter-processor interrupts (IPIs). IPIs are the only way to programmatically notify a remote processor of a local event (such as a job release) and are used to invoke the scheduler. Despite their small latencies, IPIs are not "instantaneous" and task preemptions based on IPIs incur an additional delay.

Timers and time resolution. Modern hardware platforms feature several clock devices and timers that can be used to enforce real-time requirements. While such devices typically offer high resolutions ($\leq 1\mu s$), hardware latencies and the OS's timer management overheads considerably decrease the timer resolutions available both within the kernel and at the application level [20]. Furthermore, in Linux (for the x86 architecture), high-resolution timers are commonly implemented based on per-processor devices. As some of the evaluated algorithms require timers to be programmed on *remote* processors, LITMUS$^{\mathrm{RT}}$ uses a two-step *timer transfer* operation: an IPI is sent to the remote CPU where the timer should be armed; after receiving the IPI, the remote CPU programs an appropriate local timer. Therefore, as two operations are needed to set up remote timers, scheduling algorithms that make frequent use of such timers incur higher overheads.

3 Theory, Practice, and Overheads

We now describe some of the key properties of the considered algorithms, challenges that arose when we implemented them in LITMUS$^{\mathrm{RT}}$, and how overheads can be accounted for.

Being EDF derivatives, each semi-partitioned algorithm analyzed in this paper was designed to overcome limitations of partitioned EDF (P-EDF) and global EDF (G-EDF). In each algorithm, a few tasks are allowed to migrate (like in G-EDF) and the rest are statically assigned to processors (like in P-EDF). The classification of tasks (fixed vs. migratory) and the assignment of per-processor task shares (see Sec. 2) are performed during an initial *assignment phase*.

An alternate compromise between P-EDF and G-EDF is clustered EDF (C-EDF), which partitions tasks onto clusters of cores and allows migration within a cluster. In previous studies [9, 13, 14], P-EDF proved to be very effective for HRT

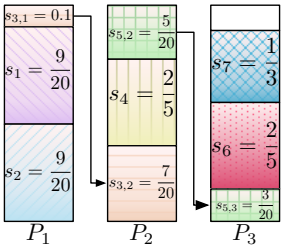

Figure 1: Example task assignment under EDF-fm.

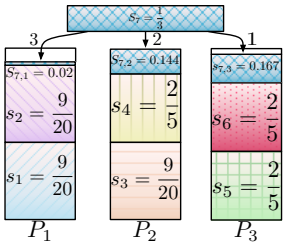

Figure 2: Example task assignment under EDF-WM.

workloads, whereas C-EDF excelled at SRT workloads. Thus, we use P-EDF and C-EDF as a basis for comparison.

3.1 EDF-fm

EDF-fm [2] was designed for SRT implicit-deadline sporadic task systems. In EDF-fm, there are at most $m-1$ migratory tasks. Each such task migrates between two specific processors, and only at job boundaries. The total utilization of migratory tasks assigned to a processor cannot exceed one, but there are no restrictions on the total system utilization. Tasks are sequentially assigned to processors using a next-fit heuristic. Suppose that T_i is the next task to be mapped and P_j is the current processor under consideration (*i.e.*, $P_1, \ldots, P_{j-1}$ have no remaining capacity). If $u_i \leq (1-c_j)$ (the capacity available on P_j), then T_i is assigned as a fixed task to P_j with a share of $s_{i,j} = u_i$. Otherwise, if $u_i > (1-c_j)$, then T_i becomes a migratory task and receives a share of $s_{i,j} = 1-c_j$ on processor P_j and $s_{i,j+1} = u_i - s_{i,j}$ on processor P_{j+1}. With this mapping strategy, at most two migratory tasks have non-zero shares on any processor. Each job of a migratory task T_i is mapped to one of T_i's assigned processors (P_j and P_{j+1}) such that, in the long run, the number of jobs of T_i that execute on P_j and P_{j+1} is proportional to the shares $s_{i,j}$ and $s_{i,j+1}$. Migratory tasks are statically prioritized over fixed tasks and jobs within each class are scheduled using EDF. With this strategy, migratory tasks cannot miss any deadlines.

Example 1. To better understand EDF-fm's task assignment phase, consider a task set τ comprised of seven tasks: $T_1 = T_2 = T_3 = (9, 20)$, $T_4 = T_5 = T_6 = (2, 5)$, and $T_7 = (1, 3)$. The total utilization of τ is $U \approx 2.88$. An assignment for τ under EDF-fm is shown in Fig. 1. In this assignment, T_3 and T_5 are the only migratory tasks. T_3 receives a share $s_{3,1} = 2/20$ on processor P_1 and $s_{3,2} = 7/20$ on processor P_2, while T_5 receives a share $s_{5,2} = 5/20$ on processor P_2 and $s_{5,3} = 3/20$ on processor P_3. In the long run, out of every nine consecutive jobs of T_3, two execute on P_1 and seven execute on P_2. This is because T_3's shares are $2/20$ and $7/20$, respectively. Job releases of T_5 are handled similarly.

3.2 EDF-WM

EDF-WM [25] was designed to support HRT sporadic task systems in which arbitrary deadlines are allowed. However, for consistency in comparing to other algorithms, we will limit attention to implicit-deadline systems. During the assignment phase of EDF-WM, tasks are partitioned among processors using a bin-packing heuristic (any reasonable heuristic can be used). When attempting to assign a given task T_i, if no single processor has sufficient available capacity to accommodate T_i, then it becomes a migratory task. Unlike in EDF-fm, such a migratory task T_i may migrate among *several* processors (not just two). However, EDF-WM aims at minimizing the number of such migratory tasks.

A migratory task T_i's per-processor shares are determined by progressively splitting its per-job execution cost e_i into "slices," effectively creating a sequence of "sub-tasks" that are assigned to distinct processors. Even though these processors may not be contiguous, for simplicity, let us denote these sub-tasks as $T_{i,k}$, where $1 \leq k \leq m'$, and their corresponding processors as $P_1, \ldots, P_{m'}$. Each sub-task $T_{i,k}$ is assigned a (relative) deadline using the rule $d_{i,k} = d_i/m'$. Each sub-task execution cost (or *slice*) $e_{i,k}$, where $1 \leq k \leq m'$, is determined in a way that minimizes the number of processors m' across which T_i is split, while ensuring that $\sum_{k=1}^{m'} e_{i,k} \geq e_i$ holds. Furthermore, assigning each $T_{i,k}$ to its processor P_k must not invalidate any deadline guarantees for tasks already assigned to P_k. Contrary to EDF-fm, the jobs of both fixed tasks and sub-tasks on each processor are scheduled using EDF (no static prioritization). The "job" of a sub-task $T_{i,k}$ cannot execute before the corresponding "job" of the previous sub-task $T_{i,k-1}$ has finished execution. To enforce this precedence constraint, EDF-WM assigns release times such that $r^j_{i,k} = r^j_{i,k-1} + d^j_{i,k-1}$, where $r^j_{i,k}$ ($r^j_{i,k-1}$) is the release time of the j-th job of $T_{i,k}$ ($T_{i,k-1}$), and $d^j_{i,k-1}$ is the relative deadline of $T_{i,k-1}$.

Example 2. Fig. 2 shows an example task assignment (using the first-fit bin-packing heuristic) for EDF-WM for the same task set of Example 1. In EDF-WM, only task $T_7 = (1, 3)$ is migratory. Each job of T_7 executes on processors P_3, P_2, and P_1 in sequence. For each sub-task $T_{7,k}$ of T_7 ($k \in \{3, 2, 1\}$), $d_{7,k} = d_7/3 = 1.0$. Assuming that the first job of the first sub-task of T_7, $T_{7,3}$, is released on processor P_3 at time 0, the first job of the sub-task $T_{7,2}$ would be released on P_2 at time 1, and that of $T_{7,1}$ at time 2 on P_1. The shares assigned to each sub-task are shown in the figure and correspond to execution times $e_{7,3} = 0.5$, $e_{7,2} = 0.43$, and $e_{7,1} = 0.07$.

3.3 NPS-F

NPS-F [11] was designed to schedule HRT implicit-deadline sporadic task systems. The algorithm employs a parameter δ that allows its utilization bound to be increased at the cost of more-frequent preemptions. In comparison to earlier algorithms [3, 10], NPS-F achieves a higher utilization bound, with a lower or comparable preemption frequency. The assignment phase for NPS-F is a two-step process. In the first step, the set of all n tasks is partitioned (using the first-fit heuristic) among as many unit-capacity *servers* as needed. (A server in this context can be viewed as a virtual uniprocessor.) Since n is finite and no tasks are split, the first step results in the creation of $\tilde{m}$ servers (for some $\tilde{m} \in \{1, \ldots, n\}$). In the second step, the capacity c_i of each server N_i is increased by means of an inflation function $I(\delta, c_i)$ to ensure schedulability, *i.e.*, a certain amount of over-provisioning is required to avoid deadline misses. The $\tilde{m}$ servers of inflated capacity $I(\delta, c_i)$ (called *notional processors of fractional capacity*—NPS-F—in [11]) are mapped onto the m physical processors of the platform. Such a mapping is feasible iff $\sum_{i=1}^{\tilde{m}} I(\delta, c_i) \leq m$.

The mapping of servers to physical processors is similar to the sequential assignment performed by EDF-fm: a server N_i is assigned to a processor P_j as long as the capacity of P_j is not exhausted. The fraction of the capacity of N_i that does not fit on P_j is assigned to P_{j+1}.[2]

During execution, each server N_i is selected to run every S time units, where S is a *time slot length* that is inversely proportional to δ and dependent on the minimum period of the task set. Whenever a server is selected for execution, it schedules (using uniprocessor EDF) the tasks assigned to it. Thus, abstractly, NPS-F is a two-level hierarchical scheduler.

As noted earlier, there exists a clustered variant of NPS-F, denoted C-NPS-F, that was designed to entirely eliminate off-chip server (and task) migrations. Contrary to NPS-F, in C-NPS-F the physical layout of the platform is already considered during the first step of the assignment phase, and therefore, off-chip server (and task) migrations can be explicitly forbidden. Compared to NPS-F, the bin-packing-related problem to be solved in C-NPS-F during the assignment phase is harder (there are additional constraints at the server and cluster level), and therefore the schedulable utilization of C-NPS-F is inferior to that of NPS-F.

Example 3. Fig. 3 illustrates the two steps of the NPS-F assignment process using the task system τ from Example 1. Inset (a) depicts the assignment of tasks to servers. $\tilde{m} = 4$ servers are sufficient to partition τ without splitting any task. Before mapping the servers to physical processors, the capacity c_i of each server N_i is inflated using the function I. Then, the servers are sequentially mapped (using their inflated capacities) onto the three physical processors P_1–P_3. As seen in the resulting mapping inset (b), N_2 is split between P_1 and P_2, while N_3 is split between P_2 and P_3.

[2]A second mapping strategy is described in [11] as well, but both yield identical schedulability bounds and, on our platform, the one considered here reduces the number of cross-socket server and task migrations (see Sec. 4).

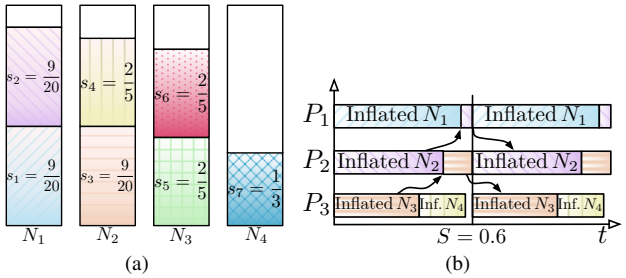

Figure 3: Example task assignment under NPS-F. The arrows in inset (b) denote that, in the first slot, N_2 first executes on P_2, then migrates to P_1; at the end of the slot, it migrates back to P_2 (similarly for N_3).

Inset (b) also shows how servers periodically execute. In this example, $S = 0.6$ (and $\delta = 4$), so every 0.6 time units, the depicted server execution pattern repeats. At time $t = S = 0.6$, server N_2 migrates from processor P_1 to processor P_2, while server N_3 migrates to processor P_3. Tasks T_3 and T_4 (assigned to N_2), and T_5 and T_6 (assigned to N_3) also migrate with their respective servers. NPS-F's mapping of servers to processors leaves only the last processor (P_3) with unallocated capacity after all servers have been mapped.

3.4 Implementation Concerns

Timing and *migration-related* problems are the major issues that need to be addressed when implementing the semi-partitioned scheduling algorithms mentioned above.

Timing concerns. In each of the algorithms above, timers are needed in order to perform various scheduling-related activities. For example, in EDF-WM, timers must be programmed to precisely enforce sub-task execution costs, and in NPS-F, timers are needed to execute servers periodically and to enforce their execution budgets. Furthermore, in both EDF-fm and EDF-WM, timers must be programmed on remote CPUs in order to guarantee that future job releases will occur on the correct processors. As noted in Sec. 2.2, programming a timer on a remote CPU entails additional costs that must be considered when checking schedulability.

A second timing concern is related to timer precision and the resolution of time available within the OS. In theory, algorithms like EDF-WM and NPS-F may reschedule tasks very frequently. For example, assuming 1 ms corresponds to one time unit, $T_{7,1}$ needs to execute for 0.07 ms in Fig. 2, while, in Fig. 3, the unused capacity (idle time) after N_4 on processor P_3 is 0.007 ms. In reality, policing such small time intervals is not possible without incurring prohibitive overheads, and reasonable minimum interval lengths must be assumed.

Migration-related concerns. In theoretical analysis, it is common to assume that job migrations take zero time. In practice, several activities (acquiring locks, making a scheduling decision, performing a context switch, *etc.*) need to be performed before a job that is currently executing on one CPU can be scheduled and executed on a different CPU. Such activities

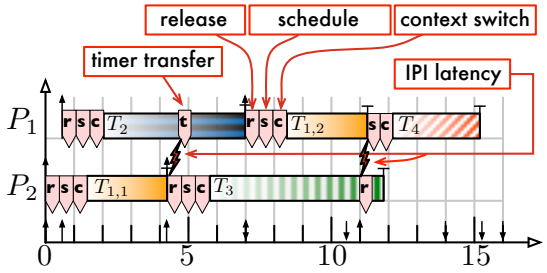

Figure 4: Example EDF-fm schedule with overheads for five jobs $T^x_{1,1} = T^{x+1}_{1,2} = (2.7, 7), T^y_2 = (5, 10), T^z_3 = (6, 11)$, and $T^w_4 = (3, 5)$ on two processors (P_1, P_2). $T^x_{1,1}$ and $T^{x+1}_{1,2}$ belong to a migratory task T_1 whose shares are assigned on P_1 and P_2. Large up-arrows denote interrupts, small up-arrows denote job releases, down-arrows denote job deadlines, T-shaped arrows denote job completions, and wedged boxes denote overheads (which are magnified for clarity). Job releases occur at $r^x_{1,1} = 0, r^{x+1}_{1,2} = r^x_{1,1} + p_1 = 7, r^y_2 = 0.5, r^z_3 = 4.2$, and $r^w_4 = 11$.

have a cost. Furthermore, given the coarse-grained protection mechanism of tasks and run queues explained in Sec. 2.2, when tasks may migrate as part of the scheduling process, extra care must be taken in order to avoid inconsistent scheduling decisions. This problem is exacerbated in scheduling algorithms such as NPS-F, where—by design—concurrent scheduling decisions are likely to happen. For example, in Fig. 3 at time $S = 0.6$, P_2 races with P_1 to schedule tasks of N_2, and (at the same time) P_3 races with P_2 to schedule tasks of N_3.

3.5 Kernel Overheads and Cache Affinity

In actual implementations, tasks are delayed by seven major sources of system overhead, five of which are illustrated in Fig 4, which depicts a schedule for EDF-fm. When a job is released, *release overhead* is incurred, which is the time needed to service the interrupt routine that is responsible for releasing jobs at the correct times. Whenever a scheduling decision is made, *scheduling overhead* is incurred while selecting the next process to execute and re-queuing the previously-scheduled process. *Context-switch overhead* is incurred while switching the execution stack and processor registers. These overhead sources occur in sequence in Fig. 4, on processor P_2 at times 0 and 4.2 when $T^x_{1,1}$ and T^z_3 are released, and again on processor P_2 at times 0.5 and 7 when T^y_2 and $T^{x+1}_{1,2}$ are released. *IPI latency* is a source of overhead that occurs when a job is released on a processor that differs from the one that will schedule it. This situation is depicted in Fig. 4, where at time 11, T^w_4 is released on P_1, which triggers a preemption on P_2 by sending an IPI. *Timer-transfer overhead* is the overhead incurred when programming a timer on a remote CPU (see Sec. 2.2). In Fig. 4, this overhead is incurred on processor P_2 at time 4.5 when the completion of the job $T^x_{1,1}$ (on processor P_1) of the migratory task T_1 triggers a request to program a timer on processor P_2 to enable the release of the next job $T^{x+1}_{1,2}$.[3] *Tick overhead* is the time needed to manage periodic scheduler-tick timer interrupts; such interrupts have limited impact under event-driven scheduling (such as EDF) and, for clarity, they are not not shown in Fig. 4. Finally, *cache-related preemption and migration delay* (CPMD) accounts for additional cache misses that a job incurs when resuming execution after a preemption or migration. The temporary increase in cache misses is caused by the perturbation of caches while the job was not scheduled.

Overhead accounting. Schedulability analysis that assumes ideal (*i.e.*, without overheads) event-driven scheduling can be extended to account for kernel overheads and CPMD by inflating task execution costs. For many of the overheads considered in this paper, standard accounting techniques exist[4] that can be applied to semi-partitioned algorithms by accounting for specific properties of these algorithms. For example, as semi-partitioned approaches distinguish between migratory and fixed tasks, migration and preemption overheads always need to be separately considered. Further, additional IPI latencies have to be accounted for in EDF-WM to reflect the operations performed to guarantee sub-task precedence constraints, and in NPS-F to ensure consistent scheduling decisions when switching between servers (which occurs when the fraction of a timeslot allocated to one server is exhausted and another server continues). In addition, NPS-F's server-switching imposes an additional overhead on all tasks executed within a server. These overheads can be accounted for by reducing the effective server capacity available to tasks.

Bin-packing. In *all* of the evaluated algorithms, problematic issues (that have not been addressed before) arise when accounting for overheads during the assignment phase. Standard bin-packing heuristics assume that item sizes (*i.e.*, task utilizations) are constant. However, when overheads are accounted for, the effective utilization of already-assigned tasks may inflate when an additional task is added to their partition (*i.e.*, bin) due to an increase in overheads. Thus, ignoring overheads when assigning tasks may cause over-utilization. To deal with this, we accounted for overheads after each task assignment and extended prior bin-packing heuristics to allow "rolling back" to the last task assignment if the current one causes over-utilization. Without these extensions, any task set exceeding the capacity of one processor would be unschedulable by the commonly-used next-fit, best-fit, and first-fit heuristics (which try to fully utilize one processor before considering others). In contrast, the worst-fit heuristic (used in [9, 13, 14, 21]) partially hides this problem since it tends to distribute unallocated capacity evenly among processors.

Considering overheads in the assignment phase of NPS-F exposes an additional issue that was not considered by the designers of that algorithm. If overheads are only accounted for *after* the mapping of servers to physical processors, then a server's allocation may grow beyond the slot length S. This would render the mapping unschedulable, as it would essen-

[3] In NPS-F, each task always executes within its server and therefore, no timer-transfer overheads are incurred.

[4] Omitted due to space constraints, *e.g.*, see [9, 13, 14, 15, 18] for details.

tially require servers and tasks to be simultaneously scheduled on two processors. However, if overheads are already accounted for during the first bin-packing phase, *i.e.*, *before* the mapping to physical processors, then it is unknown which servers (and hence tasks) will be migratory. We resolve this circular dependency by making worst-case assumptions with regard to the magnitude of overheads during the the first bin-packing phase. This approach adds pessimism, but it is required to prevent servers from becoming overloaded.

3.6 Measuring Kernel Overheads and Cache Effects

Later, in Sec. 4, we present a study conducted to compare the algorithms considered in this paper on the basis of schedulability with real overheads considered. In this section, we explain how such overheads were experimentally determined.

Our experimental platform is an Intel Xeon L7455 system, which is a 24-core 64-bit uniform memory access (UMA) machine with four physical sockets. Each socket contains six cores running at 2.13 GHz. All cores in a socket share a 12 MB L3 cache, while groups of two cores share a 3 MB L2 cache. Each core also includes two separate data/instruction L1 caches (32 KB each). Under C-EDF, we opted to group cores around L3 caches. This cluster size was selected based on guidelines given in prior studies [9, 16]. The same cluster size was also used for C-NPS-F, as it yields the highest possible utilization bound given the topology of our platform [11].

Kernel overheads. Runtime overheads were experimentally measured (with Feather-Trace [12]) using the same methodology previously employed in [9, 13, 14]. We traced workloads consisting of implicit-deadline periodic tasks under each of the six evaluated algorithms. Task set sizes ranged over [10, 350] with a granularity of 10 in the range [10, 200], and 50 in the range (200, 350].[5] For each task set size, we measured ten randomly generated task sets (with uniform light utilizations and moderate periods; see Sec. 4). Each task set was traced for 60 seconds. Due to the timing concerns mentioned in Sec. 3.4, we enforced a minimum sub-task execution cost of 50 μs under EDF-WM, and imposed a minimum server size of 150 μs under NPS-F. We further used task sets with $S \geq 2.5\ ms$ (and $\delta = 2$) to limit the number of server-switches.

In total, more than 1,300 task sets were traced, and more than 200 GB of overhead samples were collected. Average- and worst-case overheads as a function of task set size were computed for each algorithm after removing possible outliers (due to sources of unpredictability in the Linux kernel) by applying a 1.5 interquartile range (IQR) outlier filter.[6] In the full version of this paper [7], this overhead data is given in 14 graphs; due to space constraints, only worst-case release overhead, plotted in Fig. 5, is discussed here.

The most notable trends in Fig. 5 are the very high overheads

[5]These granularities allow for a higher resolution when the number of tasks is less than 200 (which is the prevalent range of task set sizes for the distributions presented in Sec. 4).

[6][28] suggests IQR as a standard technique to remove outliers.

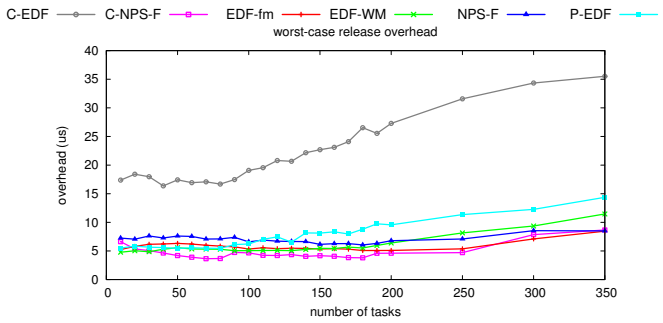

Figure 5: Sample worst-case release overheads (in μs) as function of task set size.

of C-EDF in comparison with the other algorithms, and the low overheads (within 5 μs from P-EDF) of all semi-partitioned algorithms. Under semi-partitioned approaches, migrations are *push*-based: semi-partitioned algorithms statically determine the next processor that should schedule a job (*i.e.*, the job is "pushed" to the processor where it should execute next when it finishes execution on the previous processor). Instead, under C-EDF (and under global approaches), migrations are *pull*-based: the next processor is dynamically determined at runtime (the job is "pulled" by the processor that dequeues it first from the run queue). Pull-migrations imply much higher overheads as they require global state and shared run queues that foster lock contention, which is reflected in Fig. 5.

As in [9], we used monotonic piecewise linear interpolation to determine upper bounds for each overhead (as a function of the task set size); these upper bounds were used in the schedulability experiments described in Sec. 4.

Cache effects. While kernel overheads depend on the task set size, CPMD depends on the *working set size* (WSS) of a job, and on the cache interference caused by other (possibly best-effort) jobs. In [8], we employed two empirical methodologies to assess CPMD incurred on *the same* Intel Xeon platform used in this study. Experiments were carried out in two configurations: an otherwise idle system and a system loaded with cache-polluting background tasks.

A sample of our results is shown in Fig. 6, which depicts worst-case CPMD values in a system under load (upper curves) and in an otherwise idle system (lower curves) for preemptions and each kind of migration (via L2, via L3, and via memory) as a function of WSS. In a system under load there are no substantial differences among preemption and migration costs. In contrast, in an idle system, preemptions always cause less delay than migrations, whereas L3 and memory migrations have comparable costs. In particular, if the working set fits into the L1 cache (32 KB), then preemptions are negligible (around 1μs), while they have a cost that is comparable with that of an L2 migration when the WSS approaches the size of the L2 cache. L3 and memory migrations have comparable costs, with a maximum of 3 ms for WSS = 3072 KB.

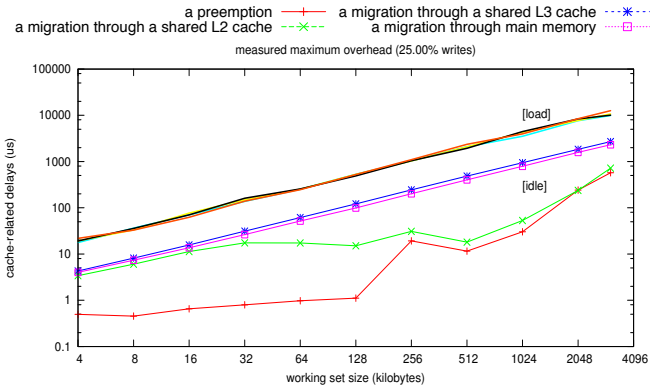

Figure 6: Worst-case CPMD (in μs) for preemptions and different types of migrations as a function of WSS (in KB). The four lower curves with point markers were measured in an idle system; the four coinciding upper curves without point markers reflect the lack of substantial differences in a system under a heavy background load.

4 Schedulability Experiments

We compared EDF-fm, EDF-WM, NPS-F, C-NPS-F, P-EDF, and C-EDF (with cores clustered at the L3 cache level) on the basis of schedulability, using an experimental setup similar to previous studies [13, 14, 17]. An algorithm's *schedulability* (HRT or SRT) is defined as the fraction of generated task sets that are schedulable (HRT or SRT) under it.

We generated implicit-deadline periodic tasks by first generating task utilizations using three uniform, three bimodal, and three exponential distributions. The ranges for the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*). For the bimodal distributions, utilizations uniformly ranged over [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*light*), 6/9 and 3/9 (*medium*), and 4/9 and 5/9 (*heavy*). For the exponential distributions, utilizations were generated with a mean of 0.10 (*light*), 0.25 (*medium*), and 0.50 (*heavy*). With exponential distributions, we discarded any points that fell outside the allowed range of $[0, 1]$. Integral task periods were then generated using three uniform distributions with ranges $[3ms, 33ms]$ (*short*), $[10ms, 100ms]$ (*moderate*), and $[50ms, 250ms]$ (*long*). The number of tasks n was determined by creating tasks until total utilization exceeded a specified cap (varied between 1 and 24, the total number of cores on our test platform) and by then discarding the last-added task.

4.1 Performance Metric

The setup described above allows schedulability to be studied as a function of an assumed utilization cap. However, when overheads are considered, the situation becomes more complex. In particular, while system overheads can be reasonably dealt with by assuming maximum (average-case) values in the HRT (SRT) case, CPMD is clearly dependent on WSS. This raises the question: *which WSS should be assumed?*

While previous studies [13, 14, 17] have focused on *selected* WSS values, in [9] we employed a methodology where CPMD becomes a parameter of the task generation procedure, thus rendering schedulability a function of two variables: an assumed cap U on total utilization and an assumed CPMD value D. This allows schedulability to be studied for a broad range of values for D, thus avoiding any bias towards a particular WSS selection. A reasonable range of values to consider for D can be determined by measuring CMPDs for various WSSs.

When this approach was applied in [9], a single CPMD value was assumed for both preemptions and the various kinds of migrations that can occur (through L2, L3, and main memory, respectively) when assessing the schedulability of a task set. Such an approach is problematic for our purposes here, as semi-partitioned algorithms are designed to lessen the impact of migrations. Thus, in this study, we express the different D values *measured on our platform* as a function of WSS. For example, considering WSS = 64 KB in an idle system, Fig. 6 tells us that a preemption has a delay $D = 1\mu s$, a migration through an L2 cache has $D = 17\mu s$, and L3 and memory migrations have $D = 60\mu s$. With such a mapping, schedulability becomes a function of the assumed utilization cap U and the assumed WSS W. In essence, one can think of W as a parameter that is used to determine an appropriate CPMD value D by indexing into either the graph in Fig. 6 or its average-case-delay counterpart.

To avoid 3D graphs for schedulability (which depends on both U and W), we adopt the *weighted schedulability* approach used in [9]. Let $S(U, W) \in [0, 1]$ denote an algorithm's schedulability for a given U and W, and let Q denote a range of utilization caps (as defined by the experimental setup). Then *weighted schedulability*, $S(W)$, is defined as $S(W) = (\sum_{U \in Q} U \cdot S(U, W))/(\sum_{U \in Q} U)$. A complete discussion of weighted schedulability can be found in [9].

4.2 Schedulability Tests

For each algorithm and each pair (U, W), we determined $S(W)$ by checking 100 task sets. We varied U from one to 24 in steps of 0.25, and W over $[0, 3072]$ KB in steps of 16 KB for $W \leq 256$ KB, in steps of 64 KB for 256 KB $< W \leq 1024$ KB, and in steps of 256 KB for higher values. This allows for a higher resolution in the rage of WSSs that have low CPMD ($D \leq 1ms$ in a system under load—Fig. 6). The upper bound of 3072 KB for W was selected because measurements taken on our test platform revealed that CPMD becomes unpredictable (over many measurements, standard deviations are large) for WSSs exceeding this bound. We used maximum (resp., average) overhead and CPMD values to determine weighted schedulability in the HRT (resp., SRT) case. For CPMD, both loaded and idle systems were considered.

The schedulability of a single task set was checked as follows. For P-EDF and C-EDF, we determined whether each task could be partitioned using the *worst-fit decreasing* heuristic. For P-EDF (C-EDF), HRT (SRT) schedulability on each processor (within each cluster) merely requires that that processor (cluster) is not over-utilized. For EDF-fm, EDF-WM,

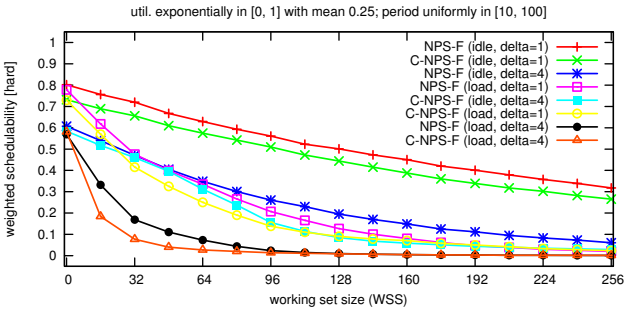

Figure 7: Comparison of NPS-F and C-NPS-F schedulability for various δ values in loaded and idle systems. (Labels ordered as the curves appear for WSS $= 96$.)

NPS-F, and C-NPS-F, we determined schedulability by using tests (SRT for EDF-fm, HRT for the others) presented by the developers of those algorithms; these tests were augmented to account for overheads, as discussed earlier.

The considered scenarios resulted in 54 graphs of weighted schedulability data arising from testing the schedulability of approximately 7 million task systems under each algorithm; when expanded to produce actual (not weighted) schedulability plots, over 1,500 graphs are required. Due to space constraints, we only discuss a few representative weighted schedulability graphs here. We further restrict our attention to $W \leq 1024$ KB because all major trends manifest in this range. All graphs (both weighted and actual schedulability, and for the full WSS range) can be found in the full version of this paper [7].

4.3 NPS-F, C-NPS-F, and Choosing δ

NPS-F and C-NPS-F actually represent a "family" of different design choices, as the behavior of each algorithm depends on the parameter δ. We begin with two observations concerning these algorithms that allow us to reasonably constrain the considered design choices for these algorithms in later graphs.

Observation 1. *$\delta = 1$ leads to higher schedulability than $\delta = 4$.* Under NPS-F (Sec. 3.3), increasing δ leads to a higher utilization bound at the cost of increased preemption frequency. In [11], Bletsas and Andersson presented a comparison of NPS-F's schedulable utilization bounds with δ ranging over $[1, 4]$. $\delta = 4$ was shown to yield a higher bound than $\delta = 1$, at the cost of increased preemptions. In contrast to this, we found that when overheads are considered, NPS-F schedulability is almost always better with $\delta = 1$ than with $\delta = 4$ in both loaded and idle systems. In particular, we found $\delta = 4$ to be competitive with $\delta = 1$ only when bin-packing issues *and* CPMD are negligible (for uniform light distributions, with small WSSs, and an idle system; see [7]). The difference can be observed in Fig. 7, which plots $S(W)$ for NPS-F for both idle systems and systems under load, for both $\delta = 1$ and $\delta = 4$. Here, NPS-F schedulability is always better with $\delta = 1$ than with $\delta = 4$. This indicates that preemption-related overheads negatively impact schedulability when $\delta = 4$. Given Obs. 1, we only consider the choice of $\delta = 1$ in the graphs that follow.

Observation 2. *C-NPS-F is almost never preferable to NPS-F.* Fig. 7 shows that C-NPS-F is never preferable to NPS-F in idle systems or in systems under load when $\delta = 1$. Eliminating off-chip migrations in C-NPS-F exacerbates bin-packing-related issues that arise when assigning servers to processors and heavily constrains C-NPS-F schedulability. Because of its poor performance in comparison to NPS-F, we do not consider C-NPS-F in the graphs that follow.

We note that in the full version of the paper [7], C-NPS-F and the choice of $\delta = 4$ are considered in all graphs.

4.4 HRT and SRT Schedulability Results

Fig. 8 gives a subset of the weighted schedulability results obtained in this study (again, all results can be found in [7]). The left column of the figure gives HRT schedulability results for the exponential medium (inset (a)) and bimodal heavy (inset (c)) distributions, while the right column reports SRT results for the exponential medium (insets (b)) and uniform heavy (inset (d)) distributions. Note that the curves for loaded and idle systems virtually coincide for every algorithm in the SRT case (insets (b,d)), whereas large differences are apparent in the HRT case (insets (a,c)). This is due to the relative magnitude of CPMD values: worst-case CPMD values are much higher in a system under load than in an idle system (Fig. 6), while the difference for average-case values is much less pronounced [8]. The following observations are supported by the data we collected in the context of this study.

Observation 3. *EDF-WM is the best performing algorithm in the HRT case.* In the HRT case, EDF-WM overcomes bin-packing-related limitations that impact P-EDF when many high-utilization tasks exist (Fig. 8(c)). More generally, EDF-WM always exhibits schedulability in this case that is superior, or at worst comparable (*e.g.*, Fig. 8(a)), to that of P-EDF.

Observation 4. *EDF-WM outperforms C-EDF in the SRT case.* Fig. 8(b) shows that, for C-EDF, schedulability decreases quickly as WSS increases due to bin-packing limitations, which are exacerbated by high(er) overheads due to higher run-queue contention. In contrast, EDF-WM exhibits good schedulability over the whole range of tested WSS values.

Observation 5. *EDF-fm usually outperforms C-EDF in the SRT case.* Like EDF-WM, EDF-fm achieves higher schedulability than C-EDF when most tasks have low utilization (*e.g.*, Fig. 8(b)). However, due to the utilization constraint EDF-fm imposes on migratory tasks, it is unable to schedule task sets where most tasks have high utilization (Fig. 8(d)).

Observation 6. *NPS-F is inferior to the other scheduling approaches in most of the analyzed scenarios.* In Fig. 8, schedulability under NPS-F is lower than that of all the other evaluated scheduling policies in all depicted scenarios (HRT and SRT). NPS-F schedulability is heavily constrained by the pessimistic assumptions made in the bin-packing heuristics of NPS-F's assignment phase, and by higher preemption and migration delays. NPS-F achieves *slightly* better schedulability results

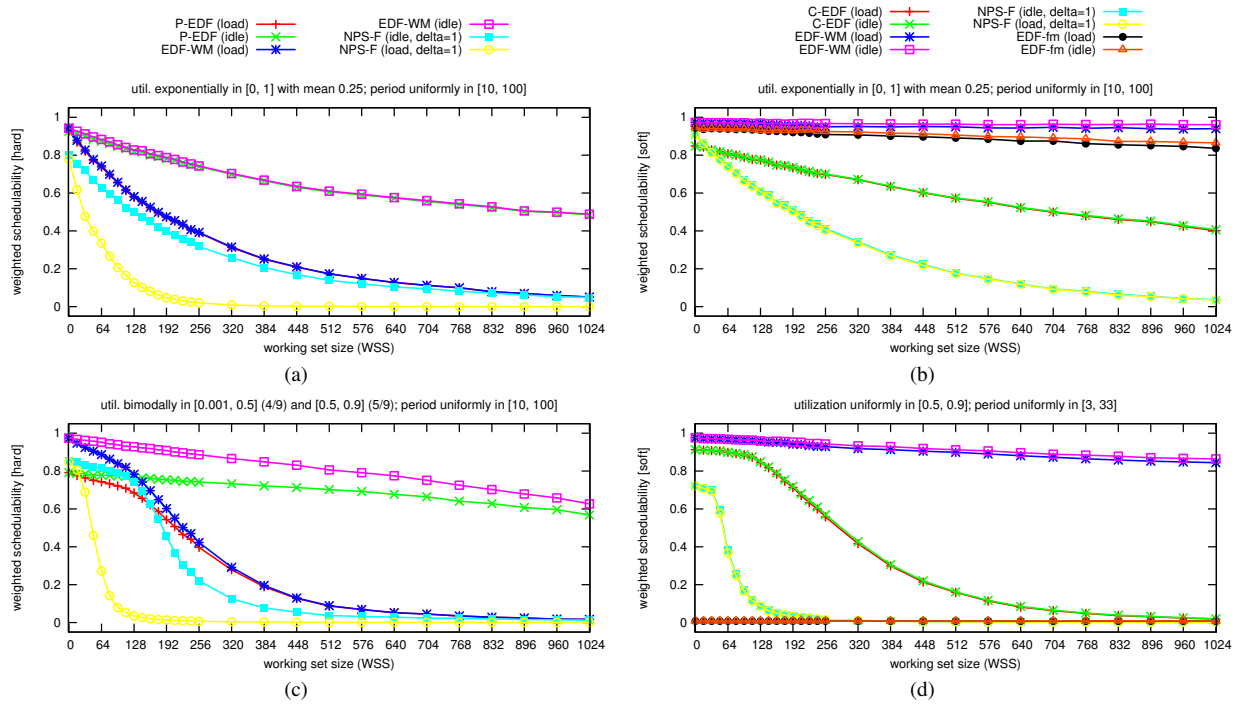

Figure 8: Weighted schedulability as a function of WSS. **(a)** HRT results for medium exponential utilizations and moderate periods. **(b)** SRT results for medium exponential utilizations and moderate periods. **(c)** HRT results for heavy bimodal utilizations and moderate periods. **(d)** SRT results for heavy uniform utilizations and short periods.

than the other algorithms only when bin-packing issues are negligible and CPMD has a limited impact (*i.e.*, in the uniform light utilization scenario, with small WSSs; see [7])

4.5 Design Principles

The observations above support the conclusion that semi-partitioning can offer benefits over conventional partitioned, global, and clustered scheduling approaches, but not all design choices in realizing a semi-partitioned approach will give good results in practice. In the following, we summarize a number of design principles that we suggest should be followed in further work on semi-partitioned scheduling. These principles are derived from the observations above and our experiences in implementing the various algorithms considered in this paper.

Avoid unneeded migrations. EDF-WM reduces to pure partitioning when task utilizations are low (no task needs to migrate). In contrast, EDF-fm and NPS-F migrate tasks even in low-utilization contexts where partitioning would have been sufficient. This increases overheads and contributes to their lower schedulability (Obs. 5 and 6).

Minimize the number of preemptions. Avoiding migrations by increasing preemption frequency can negatively impact schedulability. This was one of the issues considered in Obs. 1, where increased preemption frequency was seen to lower schedulability under NPS-F. Also, in many cases, the difference in the cost of a preemption and that of a migration through L2, L3, or memory is not significant (particularly, in a system under load, as seen in Fig. 6). Thus, favoring preemptions over migrations generally, L2 over L3 migrations, *etc.*, may not lead to improved schedulability (Obs. 2).

*Minimize the number of tasks that **may** migrate.* Migrating servers with tens of tasks—any of which could incur CPMD—increases analysis pessimism and leads to lower schedulability (Obs. 6). Higher schedulability is achieved by bounding the number of migrating tasks (Obs. 3, 4, and 5).

Avoid pull-migrations in favor of push-migrations. Push-migrations entail lower overheads than pull-migrations (Sec. 3.6). This is because push-migrations can be planned for in advance, while pull-migrations occur in a reactive way. Due to this difference, push-migrations require only mostly-local state within per-CPU run queues, while pull-migrations require global state and shared run queues. High overhead due to run-queue contention is one reason why schedulability is lower under C-EDF than under EDF-WM and EDF-fm (Obs. 4 and 5). *One of the key virtues of (most) semi-partitioned algorithms is that they enact migrations by following a pre-planned strategy; this is unlike how migrations occur under most conventional global and clusterd algorithms.*

Migration rules should be process-stack-aware. Migrations and context switches are not "instantaneous"; situations where migrating tasks are immediately eligible on another CPU (*e.g.*, at an NPS-F slot boundary) need careful process-stack management (so that each task executes on a single CPU only) that is tricky to implement and entails analysis pessimism.

Use simple migration logic. Migrating tasks at job boundaries (task migration) is preferable to migrating during job execution (job migration). Migrations of the former type en-

tail less overhead, are easier to implement, and are more predictable. This results in a much simpler admission test (*e.g.*, EDF-fm's), particularly when overheads must be considered.

Be cognizant of overheads when designing task assignment heuristics. Such heuristics are crucial for an algorithm's performance and should have an overhead-aware design to avoid excessive pessimism (Obs. 2 and 6).

Avoid two-step task assignments. With double bin-packing, pessimistic analysis assumptions concerning the second phase must be applied when analyzing the first phase. For example, when analyzing the first assignment phase of NPS-F, pessimistic accounting is needed for migrating servers, because the second phase determines which servers actually migrate.

5 Conclusion

We have presented the first empirical study of semi-partitioned real-time scheduling algorithms under consideration of real-world overheads. Our results indicate that, from a schedulability perspective, semi-partitioned scheduling is often better than other alternatives. Most importantly, semi-partitioned schedulers can benefit from the pre-planned nature of push-migrations: because it is known ahead of time *which task* will migrate, and also among *which processors*, CPMD accounting is task-specific and hence less pessimistic. Perhaps surprisingly, we found that this advantage extends even to scenarios in which cache-related costs of migrations are not substantially worse than those of preemptions: since push-migrations can be implemented with mostly-local state, kernel overheads are much lower in schedulers that avoid pull-migrations.

In future work, we would like to consider static-priority semi-partitioned algorithms, and evaluate the impacts of real-time synchronization protocols on semi-partitioned schedulers.

Acknowledgement: This study took a year and a half to complete and benefited from the feedback from a number of people; of these, Chris Kenna and Alex Mills deserve special mention.

References

[1] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Sys.*, pp. 199–208, 2005.

[2] J. Anderson, V. Bud, and U. Devi. An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems. *Real-Time Syst.*, 38:85–131, 2008.

[3] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemptions. In *Proc. of the 20th Euromicro Conf. on Real-Time Sys.*, pp. 243–252, 2008.

[4] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proc. of the 29th Real-Time Sys. Symp.*, pp. 385–394, 2008.

[5] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *Proc. of the 12th Int'l Conf. on Embedded and Real-Time Computing Sys. and Apps.*, pp. 322–334, 2006.

[6] B. Andersson, E. Tovar, and P. B. Sousa. Implementing slot-based task-splitting multiprocessor scheduling. Technical Report HURRAY-TR-100504, IPP Hurray!, May 2010.

[7] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? Extended version of this paper. Available at http:// www.cs.unc.edu/ ~anderson/papers.html.

[8] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proc. of the 6th Int'l Workshop on Operating Sys. Platforms for Embedded Real-Time Apps.*, pp. 33–44, 2010.

[9] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proc. of the 31st Real-Time Sys. Symp.*, pp. 14–24, 2010.

[10] K. Bletsas and B. Andersson. Notional processors: An approach for multiprocessor scheduling. In *Proc. of the 15th Symp. on Real-Time and Embedded Technology and Apps.*, pp. 3–12, 2009.

[11] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proc. of the 30th Real-Time Sys. Symp.*, pp. 447–456, 2009.

[12] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *In Proc. of the Third Int'l Workshop on Operating Sys. Platforms for Embedded Real-Time Apps.*, pp. 61–70, 2007.

[13] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proc. of the 30th Real-Time Sys. Symp.*, pp. 214–224, 2009.

[14] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. of the 29th Real-Time Sys. Symp.*, pp. 157–169, 2008.

[15] B. Brandenburg, H. Leontyev, and J. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *Proc. of the 15th Int'l Conf. on Embedded and Real-Time Computing Sys. and Apps.*, pp. 273–283, 2009.

[16] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proc. of the 19th Euromicro Conf. on Real-Time Sys.*, pp. 247–256, 2007.

[17] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th Real-Time Sys. Symp.*, pp. 111–123, 2006.

[18] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 2006.

[19] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In *Proc. of the 16th Real-Time and Embedded Technology and Apps. Symp.*, pp. 165–174, 2010.

[20] D. V. Hart and S. Rostedt. Internals of the RT Patch. In *In Proc. of the Linux Symp.*, pp. 161–172, 2007.

[21] S. Kato, R. Rajkumar, and Y. Ishikawa. AIRS: Supporting interactive real-time applications on multicore platforms. In *Proc. of the 22nd Euromicro Conf. on Real-Time Sys.*, pp. 47–56, 2010.

[22] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. In *Proc. of the 13th Int'l Conf. on Embedded and Real-Time Computing Sys. and Apps.*, pp. 441–450, 2007.

[23] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proc. of the 8th ACM international conference on Embedded software*, pp. 139–148, 2008.

[24] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *Int'l Symp. on Parallel and Distributed Processing*, pp. 1–12, April 2008.

[25] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proc. of the 21st Euromicro Conf. on Real-Time Sys.*, pp. 249–258, 2009.

[26] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Sys.*, 44(1):26–71, February 2010.

[27] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *Proc. of the 2010 22nd Euromicro Conf. on Real-Time Sys.*, pp. 3–13, 2010.

[28] NIST/SEMATECH. e-Handbook of Statistical Methods. http:// www.itl.nist.gov/div898/handbook/, 2010.

[29] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094–1117, 2006.

[30] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$ homepage. http://www.cs.unc.edu/ ~anderson/litmus-rt.