$\mathrm{LITMUS}^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers [*]

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

We present a real-time, Linux-based testbed called $\mathrm{LITMUS}^{\mathrm{RT}}$*, which we have developed for empirically evaluating multiprocessor real-time scheduling algorithms. We also present the results from such an evaluation, in which partitioned earliest-deadline-first (EDF) scheduling, preemptive and nonpreemptive global EDF scheduling, and two variants of the global PD*^{2} *Pfair algorithm were considered. The tested algorithms were compared based on both raw performance and schedulability (with real overheads considered) assuming either hard- or soft-real-time constraints. To our knowledge, this paper is the first attempt by anyone to compare partitioned and global real-time scheduling approaches using empirical data.*

1 Introduction

Interest in techniques for effectively scheduling real-time workloads on multiprocessors has been increasing in recent years. Two factors are driving this interest. First, algorithmic research on this topic has resulted in a number of new scheduling approaches that remove some of the fundamental barriers imposed by partitioning approaches, the schemes most commonly considered in earlier work. Second, the landscape in terms of hardware platforms has been changing: "server-class" multiprocessor machines have been available for some time now, and chip makers are shifting to multicore technologies as a solution to the "thermal roadblock" imposed by single-core designs. In multicore platforms, several processor cores are placed on a single chip. Most major chip manufacturers currently offer dual-core chips, and some designs with more cores are also available. In the coming years, chips with 32 or more cores are expected [15]. This shift in the thinking of chip makers is a watershed event, as it will fundamentally change the "standard" computing platform in many settings to be a multiprocessor.

Given this convergence of events, the time is now ripe to extend prior work on algorithmic techniques for scheduling real-time multiprocessor systems to obtain realistic implementations that facilitate empirical comparisons. In this paper, we attempt to do just this. While implementations of partitioning approaches exist, no implementations of many of the *global* real-time scheduling approaches that have been the subject of recent theoretical interest have been produced before, at least as can be found in the published literature. Global approaches differ from partitioning approaches in that, in the latter, tasks are statically assigned to processors, while in the former, they may migrate.

Global scheduling algorithms are better able than partitioning approaches to utilize multiprocessor systems when system overheads are negligible. For example, the global PD^2 Pfair algorithm can schedule on M processors any periodic task system with total utilization at most M [2], and the global earliest-deadline-first (EDF) algorithm can ensure bounded deadline tardiness for any such task system, again, if total utilization is at most M [10, 20]. In contrast, there exist task systems with total utilization of approximately $M/2$ that no partitioning approach can correctly schedule, even if bounded deadline tardiness is allowed.

While global scheduling algorithms may be theoretically superior, they tend to have higher scheduling and migration costs than partitioning schemes. As a result, many researchers have been dismissive of global algorithms from a practical standpoint. One of our main goals in this paper is to determine whether this viewpoint is warranted. In particular, we would like to know how partitioning and global real-time scheduling approaches compare when real overheads, empirically determined, are considered.

Contributions. Driven by the issues raised above, we have constructed a testbed, which we call $\mathrm{LITMUS}^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), to compare various real-time multiprocessor scheduling approaches. One of the major contributions of this paper is to describe $\mathrm{LITMUS}^{\mathrm{RT}}$ and its use. We believe that $\mathrm{LITMUS}^{\mathrm{RT}}$ may be useful to other researchers as well. $\mathrm{LITMUS}^{\mathrm{RT}}$ was implemented by modifying the Linux 2.6.9 OS kernel configured to run on a symmetric multiprocessor (SMP) architecture. (Most aspects of this paper should remain the same for any release version of Linux 2.6.) Our particular development platform is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.70 GHz, with 8K instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory.

As a second contribution, we report on results obtained using $\mathrm{LITMUS}^{\mathrm{RT}}$ on our test platform to compare various

[*]Work supported by NSF grants CNS 0309825, CNS 0408996, and CCF 0541056, and by a grant from Intel Corp. The third author was supported by an NSF fellowship, and the fourth author by an IBM fellowship.

real-time multiprocessor scheduling algorithms. Five algorithms were considered: partitioned EDF (P-EDF), preemptive and nonpreemptive global EDF (G-EDF and NG-EDF), and two variants of the global PD^2 Pfair algorithm [2]. (These algorithms are described in the next section.) The tested algorithms were compared on the basis of both raw performance and schedulability (with real overheads considered) assuming either hard- or soft-real-time constraints. Raw performance was assessed by measuring task completion times. Lower completion times are desirable in settings where good average-case performance is required in addition to worst-case predictability. We found that all tested schemes showed somewhat comparable performance (though we did note some differences, as discussed later). For schedulability with hard-real-time constraints, the two PD^2 variants and P-EDF tended to perform the best, and with soft-real-time constraints, the two PD^2 variants and the two global EDF variants tended to perform the best. These results show that, *for each tested scheme, scenarios exist in which it is a viable choice*. Further, *they call into question the belief that global approaches are not practically viable*.

We chose to create our testbed by modifying Linux instead of an existing real-time OS (RTOS) for two reasons. First, Linux is free, open-source software that is easy to obtain and modify, and is widely accepted by both developers and end users. Second, the potential client base for $\mathrm{LITMUS}^{\mathrm{RT}}$ as it evolves will mainly include real-time graphics and multimedia applications, many of which have been developed within our own department. The timing constraints in these applications are usually soft, and the developers of those produced locally actually prefer Linux as a development platform.

A few limitations of our experiments are worth noting. First, while we believe that many of our conclusions are of a general nature, these conclusions have been drawn based on empirical data taken from *one* test platform, a traditional four-processor SMP. In future work, we hope to evaluate larger platforms, and also other architectures, most significantly multicore platforms; our design of $\mathrm{LITMUS}^{\mathrm{RT}}$ should allow it to be easily ported to these other settings. Second, in creating $\mathrm{LITMUS}^{\mathrm{RT}}$, producing a fully-featured system was *not* our goal—this would simply not be feasible at the present time. Rather, our goal was to produce a platform that would suffice for the purposes of this paper. For this reason, our experiments have involved independent, static tasks. We leave issues such as support for synchronization and I/O and dynamic workloads as future work.

Related work. Most prior work on RTOSs has focused on *uniprocessor* systems—see [17] for a recent survey. In most such work, techniques for *scheduling* multiprocessor workloads are rarely discussed. The prevailing attitude seems to be that, on a multiprocessor platform, partitioning is the only viable choice, and therefore, scheduling reduces to a uniprocessor problem. Given this prior emphasis, we mostly limit our discussion of prior work to research that pertains to Linux or that addresses multiprocessor systems more directly. We do not have sufficient space to discuss all prior Linux-related development efforts, so only those of direct relevance to our work are considered.

One such effort is RTLinux [21], which runs real-time tasks in a thin real-time kernel, with Linux itself running on top of this kernel as a low-priority *background task*. This strategy prevents the Linux kernel from disrupting real-time tasks, but at the same time, restricts the ability of such tasks to invoke Linux kernel services. We have implemented $\mathrm{LITMUS}^{\mathrm{RT}}$ differently, specifically, by incorporating the scheduling algorithms that we require directly into Linux itself. RTLinux supports periodic threads, but scheduling is limited to FIFO, round-robin, and fixed-priority schemes. While various multiprocessor scheduling algorithms could potentially be incorporated within RTLinux, we chose not to do so, because this would preclude supporting in a straightforward manner tasks that require the services of the base kernel. (We hope to enhance predictability within Linux system calls in future work.)

Another relevant Linux-related prior effort is work by Abeni *et al.* [1], who measured latencies associated with timer resolution and non-preemptive sections for several Linux variants under different types of system "stress" (*e.g.*, I/O stress, memory stress, *etc.*). They found that preemptable, lock-breaking kernels using high-resolution timers are the most effective at handling these stresses. In this paper, we provide similar measurements but with an emphasis on *multiprocessor* overheads such as task migration.

To our knowledge, multiprocessor-based RTOSs were first considered as part of work on the Spring kernel [18]. The scope of Spring extended beyond stand-alone multiprocessor systems and encompassed distributed systems composed of several multiprocessing nodes and tasks with synchronization requirements. Spring predated almost all of the recent advances in multiprocessor scheduling theory that led us to construct $\mathrm{LITMUS}^{\mathrm{RT}}$.

In other recent work concerning multiprocessors, Stohr *et al.* [19] presented the RECOMS software architecture, which is a framework for running a general-purpose OS and an RTOS on the same multiprocessor machine. This framework partitions the system by placing the general-purpose OS on its own processor and preventing I/O accesses from interfering with the RTOS. RECOMS was designed as an extension to RTAI [11], which is closely related to RT-Linux, and therefore the work in [19] is different from ours in the same ways.

Organization. The rest of this paper is organized as follows. In Sec. 2, we present a brief description of the scheduling algorithms that we evaluated. Then, in Sec. 3, we describe our implementation of $\mathrm{LITMUS}^{\mathrm{RT}}$. In Sec. 4, we present the experimental results mentioned earlier, and in Sec. 5, we conclude.

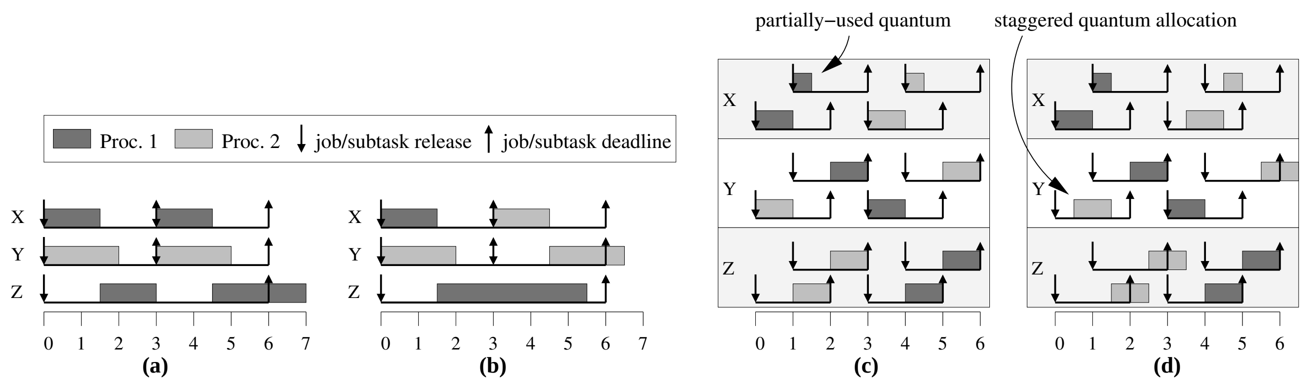


Figure 1: **(a)** G-EDF, **(b)** NG-EDF, **(c)** PD^2, and **(d)** S-PD^2 schedules of a two-processor system of three tasks: X, with an execution cost of 1.5 and period of 3.0, Y with an execution cost of 2.0 and a period of 3.0, and Z with an execution cost of 4.0 and a period of 6.0.

2 Background

We focus herein on the scheduling of *periodic task systems*. Each task in such a system is invoked or *released* repeatedly; each such invocation is called a *job* of the task. A periodic task is specified by a *period*, which denotes the (exact) separation between its successive job releases, and by an *execution cost*, which denotes the maximum execution time of any of its jobs. Each job of a task has a deadline corresponding to the release time of the task's next job. Task periods are assumed to be integral with respect to the length of the system's scheduling quantum, but execution costs may be non-integral. A task's *utilization* or *weight* is given by the ratio of its execution cost and period. As noted earlier, both EDF and Pfair scheduling algorithms are considered in this paper.

In EDF scheduling algorithms, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. As noted in the introduction, we consider three EDF variants: P-EDF, NG-EDF, and G-EDF. In P-EDF, tasks are statically assigned to processors and those on each processor are scheduled on an EDF basis. In NG-EDF, tasks may migrate, but once a job commences execution on a processor, it will run to completion on that processor without preemption. Thus, *jobs* may not migrate. Finally, G-EDF allows jobs to be preempted and permits job migration with no restrictions. No variant of EDF is optimal, *i.e.*, deadline misses can occur under each EDF variant in feasible systems (*i.e.*, systems with total utilization at most the number of processors). It has been shown, however, that deadline tardiness under NG-EDF and G-EDF is bounded in such systems [10, 20].

In Pfair scheduling algorithms [5, 16], a task T of weight $T.wt$ is scheduled one quantum at a time in a way that approximates an *ideal* allocation in which it receives $L \cdot T.wt$ time over any interval of length L. This is accomplished by sub-dividing each task into a sequence of quantum-length *subtasks*, each of which must execute within a certain time *window*, the end of which is its *deadline*. Subtasks are scheduled on an EDF basis, and tie-breaking rules are used in case of a deadline tie. A task's subtasks may execute on any processor, but not at the same time (*i.e.*, tasks must execute sequentially). The most efficient known optimal Pfair algorithm is PD^2 [2, 16], which uses two tie-breaking rules. We consider two variants of PD^2 in this paper: *synchronized* PD^2 (which we simply denote as PD^2) and *staggered* PD^2 (denoted S-PD^2) [13]. Under PD^2, quantum boundaries on different processors always align. This alignment has the potential of creating excessive bus contention at the start of each quantum, if the tasks scheduled then initially experience many cache misses in accessing memory. S-PD^2 was proposed as a solution to this problem: under it, quantum boundaries are "staggered" on different processors so that they never align. We illustrate this idea with an example below. While PD^2 is capable of ensuring all subtask deadlines for any feasible system, such deadlines can be missed under S-PD^2 by up to one quantum. This amount, though, is still considerably less than the amount by which deadlines can be missed under G-EDF and NG-EDF [10, 20]. Moreover, misses of job deadlines can be avoided in S-PD^2 by simply reducing a task's period by one quantum. Under both Pfair schemes, if a task is allocated a quantum when it requires less execution time, the unused portion of that quantum is "wasted." In contrast, under the EDF schemes considered above, such a task would relinquish its assigned quantum "early," allowing another task to be scheduled.

To see some of the differences in these algorithms, consider Fig. 1, which depicts various two-processor schedules for a system of three tasks, X, Y, and Z, as defined in the figure's caption. There are several things worth noting here. First, these three tasks cannot be partitioned onto two processors, so this system is not schedulable under P-EDF (so we do not depict a schedule for this case). Second, under each of G-EDF, NG-EDF, and S-PD^2, a deadline is missed. Third, in the NG-EDF schedule in inset (b), task Y's second job cannot execute at time 3 since Z's job must execute non-preemptively (there is actually a deadline tie here). Fourth, each task has the same window structure in insets (c) and (d). For tasks Y and Z, this is easily explained: a task's window structure is determined by its weight and both of these tasks have a weight of 2/3. As for task X, under each Pfair variant, windows are defined by assuming that each task's execution cost is an integral number of quanta. Thus, we must round up X's cost to 2.0, giving it a weight of 2/3. Because of this, some quanta allocated to task X are only half-used. Finally, note that in inset (d), quanta on processor 1 always begin at integral time instants, while on processor 2, they begin at the midpoint between two integral time instants.

3 LITMUS$^{\rm RT}$ Implementation

Our implementation efforts in developing LITMUS$^{\rm RT}$ focused on two key tasks: devising support for different quanta alignments, and incorporating the scheduling algorithms we require into Linux. These efforts are described in greater detail below. Due to space constraints, we have omitted certain details in this discussion—we plan to release a technical report soon that describes LITMUS$^{\rm RT}$ in greater depth.

3.1 Supporting Scheduling Quanta

We first discuss our methods for supporting in Linux *aligned* and *staggered* quanta, as required by PD^2 and $\mathsf{S\text{-}PD}^2$, respectively. Before describing how we accomplished this, we first digress to provide a brief introduction to the local timer interrupt hardware on our test platform and its operation in Linux. (This overview is based heavily on material from [7].)

Introduction to local timers. In our hardware configuration, each processor contains an Advanced Programmable Interrupt Controller (APIC), which is on the same chip as the processor itself. Each APIC contains a *local timer* that generates *local timer interrupts* on each processor. In Linux, a check for tasks to be scheduled is made in the local timer interrupt handlers. (Such checks are also made when a new process is created or an executing process exits, blocks, yields, or is suspended, and when returning from interrupt handlers if the execution of the handler resulted in one or more processes becoming ready.) To support time sharing, a quantum size, which can be expressed as a multiple of the period between timer interrupts, is chosen, and ready tasks of the same priority are scheduled in a round-robin fashion. As each APIC is programmed to generate interrupts at the same frequency on all processors, the interval between timer interrupts is identical across all processors. However, these interrupts do not necessarily coincide. Creating such an alignment would require that all local timers be *started* at the same time. In Linux, this is not guaranteed, since the time at which each processor starts its local timer is not predictable.

Supporting aligned quanta. We supported aligned quanta in PD^2 by making scheduling decisions only at timer interrupts, and by aligning such interrupts across processors as follows.

- After initializing local timer interrupts normally at system boot, each processor begins recording the times at which its interrupt handler is being invoked by recording the value of the Time Stamp Counter (TSC), a cycle-based 64-bit counter that records system uptime in ns.

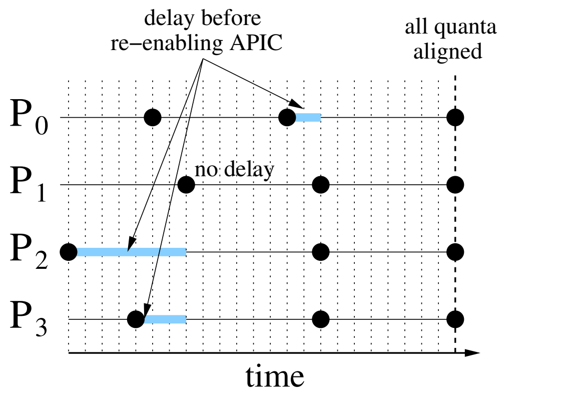


Figure 2: Illustration of the modification to support aligned quanta.

- All processors then use the TSC measurements to calculate how misaligned they are with respect to processor 0, and by how much they need to delay to align themselves.
- Each processor then disables and resets its local APIC timer, so that when it is re-enabled, it will generate its next interrupt after a full timer period.
- Finally, each processor delays appropriately, and then re-enables its local APIC timer. Processor 0 delays if it is not the most "behind" to prevent the calculation of negative delay values.

The method used to calculate the needed per-processor delays is described in [8]. Such delays were realized using a non-timer-based kernel delay function called *udelay*, which is implemented using a software loop with μs granularity. As shown in Fig. 2, we can get aligned quanta even if quanta were substantially misaligned before delaying. Such a statement cannot be made about standard Linux. Also, note that other (non-timer) interrupts cannot interfere significantly with delay times since the network and most I/O devices are not yet initialized. (It is worth noting that this approach was devised after considering many that did not work, including approaches that use *global* timer interrupts and various proposed patches.)

We also used the method described above to align timer interrupts (but not necessarily quanta) in the EDF variants we implemented. This was done in order to give each processor a consistent view of time, which is convenient when all tasks have periods that are some multiple of the quantum size. (In our implementations, both the quantum size and the period between timer interrupts were 1 ms.) However, in these EDF variants, scheduling decisions (and hence quantum allocations) do *not* always occur at timer interrupts, as is the case with PD^2. For example, if a job J in an EDF scheme completes between timer interrupts, then a new job J' may be scheduled. In our implementation, such a job J' can be preempted at the next timer interrupt, if a higher-priority job is released at that time. In such a case, J' would have executed for less than a full quantum prior to its preemption.

Supporting staggered quanta We supported staggered quanta, as required by S-PD^2, by simply adjusting the delays discussed earlier so that quantum boundaries on different processors are evenly distributed over time. For example, with a 1-ms quantum size and four processors, some processor (ideally) reaches a quantum boundary every 250 μs.

3.2 Supporting Scheduling Algorithms

Our framework for supporting multiprocessor scheduling algorithms in Linux is plugin-based, which simplifies the task of implementing different algorithms and makes the system easily extensible. Each scheduling algorithm is implemented as a plugin component. A component is specified by a collection of function pointers, which reference the functions that define the behavior of the implemented algorithm. There are three such functions: *initialization*, which installs the component into the scheduler at boot time; *tick handler*, which is called every timer interrupt; and *decision*, which makes scheduling decisions. These functions are described in greater detail below.

Before continuing, we introduce some relevant terminology. A *ready queue* is a priority-ordered queue of real-time tasks that are ready for execution, and is implemented as a linked list. A *release queue* is a queue of queues. All tasks in each such queue have the same release time, and these queues are ordered in the release queue by the release time of the tasks they contain (earliest to latest). The *Linux runqueue* is a complex per-CPU Linux data structure that maintains all Linux tasks assigned to that CPU. This structure contains pointers to both the currently-running task and the idle task for its associated CPU, as well as an *active queue*, which is a priority queue containing tasks that are ready for execution. The organization of these queues allows the scheduler to determine in constant time which task should run next when a scheduling decision needs to be made. (There is also an *expired queue* containing tasks that have exhausted their allocated time slice; however, knowledge of this queue is not necessary to understand our implementation.) In global algorithms, we used FIFO queue locks when accessing shared scheduler data structures, to ensure predictability.

Initialization function. This function is called during system boot. A kernel boot option determines which scheduler to run, and therefore which initialization function to call. This function installs its component into the scheduler by changing several function pointers to reference the functions of the component. We modified the Linux scheduler to call the functions referenced by these pointers as needed. This function also creates the ready and release queues (one per processor in the case of P-EDF).

Tick handler function. This handler, which is called on a CPU at every local timer interrupt, performs two scheduling-related activities. First, the tasks in the first queue of the release queue are merged with the ready queue, if their release time has been reached. Second, if a scheduling decision is required, then the native Linux scheduler is called. In the Pfair algorithms, this happens at every timer interrupt, while in the EDF algorithms, it happens at a timer interrupt only if new jobs are released.

Decision function. The decision function is called within the native Linux scheduler, which is called whenever a scheduling decision needs to be made. As discussed earlier, this can occur both at timer interrupts and between them. The decision function schedules the highest-priority task in the ready queue by placing it in the active queue of the Linux runqueue structure for the current processor with a priority higher than any other task in that queue. As a result, the native per-processor Linux scheduler switches to this task. Note that this implementation also supports the preemption of real-time tasks. A preemption will cause the currently-executing real-time task to be removed from the active queue and returned to the ready queue.

Example. As a concrete example, we briefly describe the implementation of one of our scheduling algorithms, NG-EDF, pseudo-code for which is given in Fig. 3. The plugin component for NG-EDF maintains a global ready queue in EDF priority order, and a global release queue. The tick handling function is called at every quantum boundary and decreases the budget of the current job. Because jobs cannot preempt each other, the rescheduling function is called only when the current job exhausts its execution budget. The decision function is fairly straightforward: it simply places into the active queue of the Linux runqueue structure the task that has the earliest deadline, which is at the head of the ready queue. The other algorithms are implemented similarly, with only a few differences: for example, PD^2 does not permit scheduling decisions between quantum boundaries.

User API. The system operates in one of two modes: real-time or non-real-time. It boots in non-real-time mode (during which it initializes the appropriate real-time plugin). In order to run a real-time task set, that task set must first be created in non-real-time mode, and then the system must be switched to real-time mode to begin execution. A user performs these activities by invoking several system calls to create tasks, set task parameters, and prepare tasks for execution. When the execution of a real-time task set is complete, or the user wishes to end real-time task execution, the system can be switched back to non-real-time mode.

As noted earlier, real-time tasks in LITMUS$^{\mathrm{RT}}$ must be statically-defined and cannot invoke synchronization mechanisms. They also cannot invoke system calls or perform I/O operations, as their timing is unpredictable. (Note that this

```
SchedulerTick()
1 $CurrentTask.Budget := CurrentTask.Budget - 1$;
2 if $CurrentTask.Budget \leq 0$ then
      ▷ Replenish execution budget
3     $CurrentTask.Budget := CurrentTask.ExecCost$;
4     SetNeedResched($CurrentTask$);
      ▷ Mark that a new job must be released
5     SetNewRelease($CurrentTask$)
  fi
6 AcquireLock($QueueLock$);
  ▷ Only one processor effectively merges queues
7 Merge($ReadyQueue, ReleaseQueue.Head, CurrentTime$);
8 ReleaseLock($QueueLock$);
9 LinuxSchedulerTick()
```

```
MakeSchedDecision($CurrentTask$, CPU runqueue)
1 AcquireLock($QueueLock$);
  ▷ Unlink from local Linux runqueue
2 Unlink(CPU runqueue, $CurrentTask$);
  ▷ Examine task's state and flags
3 if ($\neg$DeadOrZombie($CurrentTask$)) then
      ▷ Reschedule is called only when a job exhausts its exec. budget
4     ReleaseNext($CurrentTask$)
  fi
  ▷ Select the highest priority real-time task
  ▷ from the ready queue
5 $next$ := ExtractMaxPrio($ReadyQueue$);
6 if $next \neq$ NIL then
7     SetHigestLinuxPriority($next$);
      ▷ Linux scheduler will select this task
8     InsertWithMaxPrio(CPU runqueue, $next$)
  fi
9 ReleaseLock($QueueLock$)
```

Figure 3: Pseudo-code for NG-EDF scheduler functions. (For simplicity, task execution costs are assumed to be integral.)

prevents the use of dynamic libraries.) Since I/O operations are not permissible, paging is not allowed in real-time mode. Hence, we lock all pages in memory and restrict all real-time tasks to share the same address space and have a relatively small memory footprint.

4 Experimental Results

In this section, we report on the results of experiments conducted using LITMUS$^{\mathrm{RT}}$ to compare the multiprocessor real-time scheduling algorithms introduced previously. We compared these algorithms on the basis of both schedulability and raw performance. The results of these experiments are presented in Secs. 4.2 and 4.3, respectively. In the schedulability evaluation, random task sets were generated and their schedulability under each scheme checked. In this evaluation, real overheads, as measured using LITMUS$^{\mathrm{RT}}$, were assumed when checking schedulability. In Sec. 4.1 below, we discuss the micro-benchmarks that were used to determine these overheads. For all experiments, measurements were taken with Linux booted into *single user mode* (*i.e.*, runlevel 1). This minimizes the impact of interrupts and other background activities by running only a minimal set of tasks that are required for a single user that is physically present (*i.e.*, not accessing the machine remotely).

4.1 Micro-Benchmarks

We measured four sources of overhead of relevance to each algorithm: task *preemption* and *migration* costs, and *context-switching* and *scheduling* overhead. Preemption and migration costs are dominated by the time it takes to reload data into a cold cache, and potentially invalidate data in remote caches. Context-switching overhead reflects the actual cost of switching between two tasks in Linux, and does not include any task-specific cache-related costs. Scheduling overhead reflects the cost of making one scheduling decision. We also measured the quantum *alignment error*, which is important for PD^2, as it strongly influences how close to optimal it is in practice. For all experiments, we measured time using the TSC, mentioned in Sec. 3.1.

Preemption and migration costs. Under each algorithm except S-PD^2, aligned quanta represent the worst-case scenario in terms of bus contention. For these schemes, such an alignment will occur at least once per hyperperiod. Thus, we measured preemption and migration costs for each scheme except S-PD^2 by focusing on one 1-ms quantum of execution on each processor, with all quanta beginning at the same time. For S-PD^2, we considered a similar situation, except that these quanta are staggered to (ideally) begin 250 μs apart, with each commencing at a different time.

Preemption costs were measured by reading the TSC before and after writing some amount of data to main memory assuming a cold cache, in the worst-case scenario where all processors are experiencing a preemption within the quantum being considered. We then subtracted from this value the time it takes to write the same amount of data in the same scenario, but with all written data being locally cached. This emulates the situation where the memory words being written were in cache prior to a preemption, but were evicted during the preemption, and thus writes after the preemption cause these words to be reloaded. The number of words to reload depends on the task's working set (WS). WS sizes (WSSs) were varied over $\{4\text{K}, 32\text{K}, 64\text{K}, 128\text{K}, 256\text{K}\}$ bytes in the micro-benchmark results. However, due to space constraints, only the 4K, 128K, and 256K cases are shown in the schedulability results presented in Sec. 4.2. (The 32K and 64K cases show similar trends to the 4K case.)

While these WSSs may appear to be small, note the following. First, we define WSS with respect to a *single quantum of computation*, and it is not possible to write too much more than 256K bytes within one 1-ms quantum. While we could have chosen to measure preemption and migration costs (see below) over several quanta for a larger WS, doing so would not be straightforward, and we believe that the

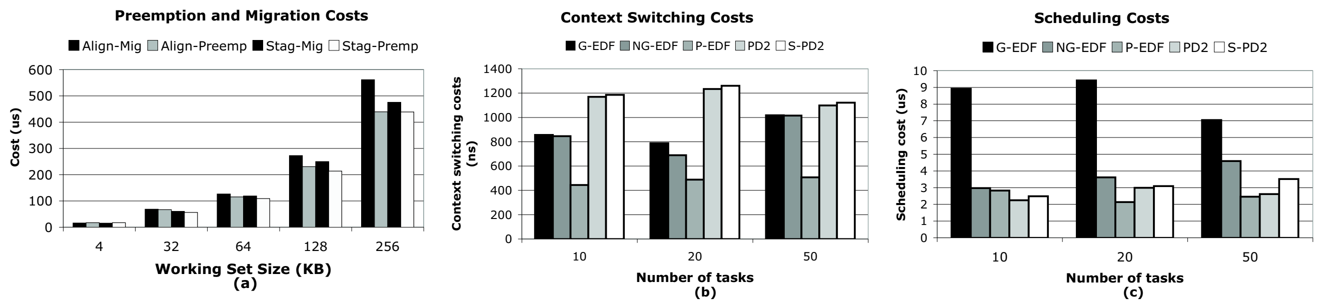


Figure 4: **(a)** Preemption/migration costs (in μs) by WSS; and **(b)** & **(c)** context-switching/scheduling costs (in ns/μs) by task-set size.

difference between these two measurement methods would be insignificant, as our method accounts for the worst-case costs that could be incurred during any quantum. Second, these WSSs are intended to be a measure of *cache reuse*, and not the total memory usage of a task during a quantum—if after a preemption or migration a task accesses memory that was never in the cache, then this access is part of the task's execution cost and does not contribute to preemption or migration costs. Many applications have high locality, so their WSS is small (in terms of reuse), even if the amount of data they access in a quantum is large. Third, due to the time-sensitive nature of real-time tasks, it may be more likely that such a task will work with a relatively small region of memory in a quantum. For all of these reasons, we believe that the smallest WSSs we have considered actually represent a fairly common case. Nonetheless, we have included larger WSSs as well, so we can assess performance as the system is stressed to its limits.

Migration costs were measured similarly, except that, when we emulated the migration of data from one cache to another, we also included the cost of invalidating that data at another processor. The cost of this invalidation can be significant, especially during a write, due to the synchronous nature of certain cache snooping protocols, and therefore we cannot simply assume that preemption and migration costs are equivalent. We forced invalidations to occur by first reading the data to be written into the private cache of another processor and by then measuring the cost of writing that data, as was done when assessing preemption costs.

Results from experiments conducted to measure both of these costs are shown in Fig. 4(a). Note that, for both the aligned and staggered cases, migration costs are higher than preemption costs, and both scale linearly with WSS, as expected. Note also that preemption costs are nearly identical for both aligned and staggered quanta for each WSS; however, migration costs are higher with aligned quanta. This result implies that cache invalidations entail less cost with staggered quanta than aligned quanta. This result makes sense, since invalidation protocols create bus traffic, and hence the likelihood of bus contention is greater when all four processors are forcing invalidations at the same time.

The methodology above attempts to estimate *worst-case* preemption and migration costs, rather than *average-case* costs. This is why we assumed that every memory access is a write, as writes are more costly than reads. Given that soft-real-time systems are a major focus of our work, we acknowledge that it is vital for us to further investigate the (non-trivial) issue of determining suitable average-case measurements for various classes of applications. Still, our measurements are valid when used to evaluate the *relative* performance of the tested scheduling algorithms. In particular, we repeated the same experiments with reads instead of writes, and found that all costs were roughly halved. Thus, we would expect the relationship between preemption and migration costs, and costs associated with aligned and staggered quanta, to remain proportionally the same when comparing some other "average-case" memory access patterns.

Context-switching overhead. We measured context-switching overhead by reading the TSC before and after a context-switch call assuming all processors perform a context switch at the beginning of a quantum that is either aligned or staggered as discussed above. The results of our measurements are shown in Fig. 4(b). Note that context-switching overhead is lowest for P-EDF, probably due to the fact that tasks do not migrate. Also, some variation among the algorithms probably arises due to differences in the time taken to load into cache the process control block of the task being switched to. For all algorithms, this overhead was several orders of magnitude lower than all other costs, and thus is relatively negligible.[1]

Scheduling overhead. We measured scheduling overhead for each algorithm by reading the TSC at the beginning and end of the code segment in which the scheduling algorithm is implemented. Results for varying task-set sizes are shown in Fig. 4(c). Overall, scheduling overheads are small (approximately 3 μs), with the exception of G-EDF. This implies that our G-EDF implementation was significantly less efficient than the others. This is due to the fact that the tick handler of G-EDF requires the lookup of M queue entries in the worst case, where M is the number of processors, whereas all other

[1]If tasks do not share an address space as assumed, then some additional overhead may be incurred to invalidate and repopulate the TLB, and to load into cache the page-table entries of the task that is being switched to.

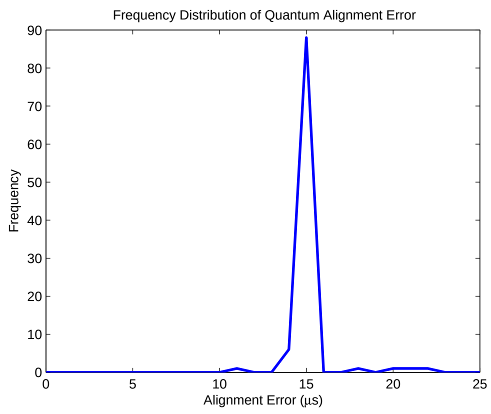


Figure 5: Quantum alignment error in LITMUS$^{\mathrm{RT}}$.

methods require only one lookup. (Our G-EDF implementation could possibly be improved, but scheduling costs are a fairly minor overhead anyway.)

Quantum alignment error. Fig. 5 shows how well quanta are aligned in LITMUS$^{\mathrm{RT}}$. These results were obtained by measuring the time between the first and last invocation of a particular timer interrupt across all processors, specifically, by reading the TSC at the beginning of each invocation of the local timer interrupt handler. Overall, 100 interrupt invocations were measured after Linux had fully booted and stabilized. More measurements were not taken due to constraints on the sizes of *proc* files, to which data was written. However, we have yet to observe an instance where the quantum alignment error changed significantly over time after the system had booted and stabilized. As seen, alignment error never exceeds approximately 25 μs with our method.

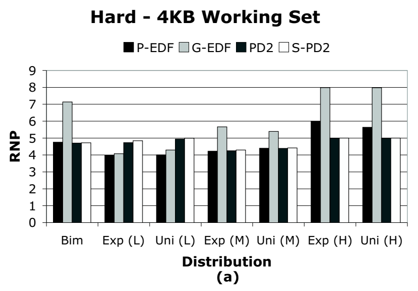
4.2 Comparison of Schedulability

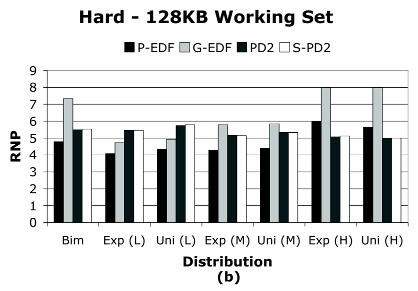
To assess differences in schedulability, we determined the schedulability of randomly-generated task sets under each scheme, for both hard- and soft-real-time systems, while varying per-task WSSs, using the overheads computed in Sec. 4.1. We used the uniform, exponential, and bimodal distributions to generate task sets as proposed by Baker [3]. (These distributions allow task-set parameters that are in keeping with multiprocessor real-time systems considered in our prior work [6].) Task periods were uniformly distributed over $[10, 100]$ (all units are in ms). Task utilizations were distributed differently for each experiment: **(i)** uniformly, over the range $[0.001, 0.1]$, $[0.1, 0.4]$, or $[0.5, 0.9]$; **(ii)** exponentially, with average 0.05 (range $[0.001, 0.1]$), 0.25 (range $[0.1, 0.4]$), or 0.7 (range $[0.5, 0.9]$), truncated as needed to achieve the desired ranges; or **(iii)** bimodally, distributed uniformly over $[0.001, 0.5)$ with probability $8/9$, and over $[0.5, 0.999]$ with probability 1/9. Task execution costs were calculated from periods and utilizations (and may be non-integral). Each task set was created by generating tasks until either total utilization was at least four (the number of processors) or a predetermined cap on the number of tasks was reached, and by then reducing the last task's utilization, if necessary, so that total utilization was exactly four. Each experiment consisted of 100 different task sets.

The definition of a *correct schedule* depends on whether we require hard- or soft-real-time guarantees. For hard-real-time systems, correctness requires that all deadlines be met, while for soft-real-time systems, it requires that deadline tardiness be bounded (regardless of how high the bound may be). We assessed differences in schedulability for each (100-task) experiment by computing the *average minimum required number of processors* (RNP) for producing a correct schedule, and the *average deadline tardiness* (TD). We also conducted an additional set of experiments in which the *total number of schedulable task sets* (NST) was determined as total utilization ranged from 2.0 to 3.9.

Before continuing, there are a few issues worth addressing. First, in the context of hard-real-time systems, we omit NG-EDF because there is currently no good hard-real-time schedulability test for it. Second, when determining schedulability under each algorithm, we first inflated the execution cost of each task to account for the overheads discussed in Sec. 4.1 using standard techniques. These techniques are described at length in [9]. (Note that, even when RNP exceeds four, we still only consider overheads as computed on a four-processor testbed. This is perhaps one limitation of our experimental methodology.) Third, when considering S-PD^2 in hard-real-time systems, task periods were reduced by one quantum, to compensate for the fact that deadlines under S-PD^2 can be missed by this amount. Finally, we determined whether a task set could be scheduled on M processors as follows. For G-EDF, the sufficient schedulability test in [4, 12] was used. For P-EDF, we determined whether each task set could be partitioned using the *first-fit decreasing* heuristic. (While a closed-form test is available for P-EDF [14], our approach is less pessimistic.) For both Pfair schemes, we simply checked if total utilization, including overheads, is at most M. Note that, in these schemes, execution costs must be rounded up to integral values after overheads are included. In the context of soft-real-time systems, schedulability under PD^2 and P-EDF was checked in the same way. However, because G-EDF and NG-EDF can guarantee bounded deadline tardiness if the system is not overloaded, only a check that total utilization is at most M is required. Finally, S-PD^2 was dealt with as in hard-real-time systems, except that task periods do not have to be decreased.

RNP for hard-real-time task sets. Fig. 6 shows RNP results for hard-real-time task sets for different task WSSs. There are several things to note here. First, schedulability under PD^2 is strongly related to WSS, because preemption and migration costs are higher in PD^2. (Note that, from Sec. 4.1, such costs are directly related to WSS.) For task sets comprised of tasks with utilizations in the range $[0.1, 0.4]$ at the smallest WSS, PD^2 has approximately the same RNP as





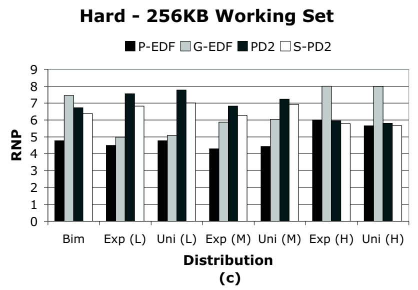
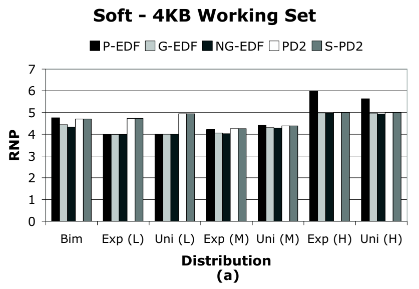
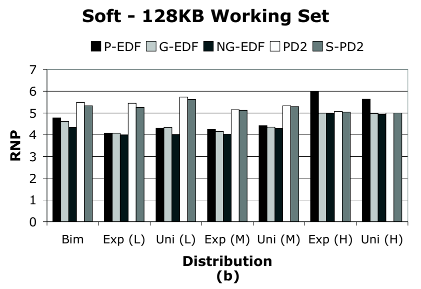


Figure 6: RNP results when scheduling hard-real-time task sets under P-EDF, G-EDF, PD^2, and S-PD^2, where WSS is **(a)** 4 KB, **(b)** 128 KB, and **(c)** 256 KB. In this and later figures, L, M, and H denote task-utilization ranges of $[0.001, 0.1]$, $[0.1, 0.4]$, and $[0.5, 0.9]$, respectively.





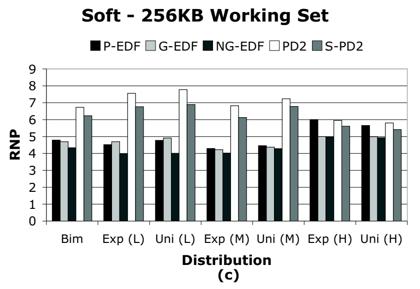


Figure 7: RNP results when scheduling soft-real-time task sets under P-EDF, G-EDF, NG-EDF, PD^2, and S-PD^2, where WSS is **(a)** 4 KB, **(b)** 128 KB, and **(c)** 256 KB.

P-EDF and a substantially lower RNP than G-EDF; however, for the same task sets at the largest WSS, PD^2 has a substantially larger RNP than both P-EDF and G-EDF. Second, schedulability under PD^2 is strongly related to task utilizations. For tasks sets comprised of very "light" tasks, P-EDF and G-EDF always outperform PD^2, whereas for task sets comprised of "heavy" tasks, PD^2 outperforms both G-EDF and P-EDF (except for when the WSS is 256 KB, in which case the RNP for PD^2 is approximately the same as G-EDF). (The terms "light" and "heavy" are meant to refer to task utilizations; *light* implies a weight of less than 1/2, and *heavy*, at least 1/2.) The reason for this behavior is that for task sets with light tasks, PD^2 incurs a significant penalty due to partially wasting quanta as a result of rounding up execution costs.[2] On the other hand, the reason why PD^2 outperforms both P-EDF and G-EDF (or is approximately the same as P-EDF) for task systems comprised entirely of heavy tasks is because heavy tasks often require fewer preemptions and migrations in PD^2 than light tasks, and under P-EDF and G-EDF many processors may be only partially utilized (due to connections to bin-packing in scheduling analysis). Third, P-EDF outperforms G-EDF for every task set. This may be because the schedulability test for P-EDF is tighter than that for G-EDF. Fourth, P-EDF substantially outperforms both G-EDF and PD^2 for systems with light tasks and large WSSs. This is because lighter tasks are easier to partition, and P-EDF does not incur the larger migration costs associated with very light tasks. Fifth, since S-PD^2 has lower migration costs, for larger WSSs, S-PD^2 outperforms PD^2; however, for smaller WSSs, PD^2 slightly outperforms S-PD^2 since S-PD^2 must decrease the period of each task by one to prevent deadline misses.

[2]One way to alleviate this problem is by choosing a smaller quantum size. To assess the impact of this, we re-ran these experiments with a 250-μs quantum. This actually worsened the performance of PD^2, because the resulting increases in preemptions and migrations negated any performance gains due to a smaller quantum size. These experiments have been omitted here due to space constraints.

RNP for soft-real-time task sets. Fig. 7 shows RNP results for soft-real-time task sets. Again, there are several things to observe. First, P-EDF and PD^2 perform about the same as before, since the same schedulability test is used for them for both hard- and soft-real-time task sets. Second, the performance of G-EDF is substantially better than before, because a restrictive schedulability test is not required here. Third, NG-EDF (not considered earlier) outperforms every other method. This behavior occurs because NG-EDF incurs no intra-job preemption or migration costs. Finally, as earlier, S-PD^2 outperforms PD^2 for large WSSs.

TD for soft-real-time task sets. Fig. 8 shows TD results for soft-real-time task sets with a WSS of 4 KB under G-EDF and NG-EDF. (We found that TD results do not change substantially as WSS changes, so we only consider the 4 KB case here, due to space constraints. In addition, tardiness is potentially unbounded under P-EDF and is zero or one quantum under the other remaining schemes, so they are also not considered.) Note first that, since task periods uniformly range over $[10, 100]$, the largest average TD observed was approximately 2.1 times the average task period. Second, TD in NG-EDF is larger than in G-EDF. This is because of priority inversions that can occur in NG-EDF due to non-preemptive execution. (Thus, non-preemptive execution has both positive and negative consequences.) Third, TD in both G-EDF and NG-EDF increases substantially as task weights increase. This is a schedulability issue: these schemes are more likely to cause deadline misses as weights increase. (As before, this is due to connections to bin-packing in scheduling analysis.)

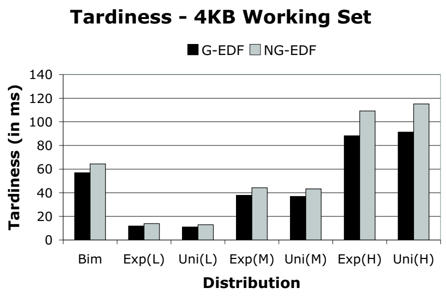


Figure 8: TD for G-EDF and NG-EDF.

NST **for hard- and soft-real-time task systems.** For both hard- and soft-real-time systems, an additional set of experiments was conducted to assess the ability of the different algorithms to meet every deadline and guarantee bounded tardiness, respectively, for arbitrary task systems on four processors. For this purpose, task sets were generated randomly with *total base utilization* (*i.e.*, total utilization before accounting for overheads) ranging from 2.0 to 3.9. Per-task utilizations were distributed either uniformly or bimodally. In the uniform case, task utilizations ranged over either $[0.1, 0.5)$ or $[0.5, 0.9)$; in the bimodal case, task utilizations were uniformly distributed in the same ranges with probabilities 0.2 and 0.8, respectively. Experiments were conducted for various WSSs for both hard- and soft-real-time systems, and in each experiment, NST values were determined for each algorithm. Results for WSSs of 128K and 4K are shown in Figs. 9 and 10 for hard- and soft-real-time systems, respectively. Additional results can be found in [9].

Several aspects of the results for hard-real-time systems in Fig. 9 are worth noting. First, the performance of G-EDF is poor for both light and heavy task systems and for both small and large WSSs. Second, unlike PD^2 and S-PD^2, there is no dramatic improvement in the performance of G-EDF and P-EDF as WSS decreases (compare the top and bottom plots). These two trends are due to connections to bin-packing in the scheduling analysis of G-EDF and in the task partitioning of P-EDF. Third, though P-EDF performs remarkably well when all tasks are light (insets (a) and (d)), the Pfair algorithms perform significantly better than P-EDF in all other cases. Also, for the Pfair algorithms, the improvement in schedulability with decreasing WSS is higher for light tasks. This is because light tasks are more significantly impacted by migration costs, as such tasks tend to be preempted (and hence migrate) more frequently.

Turning now to Fig. 10, several trends in the soft-real-time case are worth noting. First, the curves for P-EDF and PD^2 are identical to those for the hard-real-time case in Fig. 9 (for the same reason that their RNP results were similar, as discussed earlier). Similarly, the curves for S-PD^2 are nearly identical to their counterparts in Fig. 9. Second, insets (a) and (d) show that when tasks are light, schedulability is comparable for the three EDF-based algorithms when the total base utilization is at most 3.5 for both tested WSSs. With the larger WSS (inset (a)), as with the RNP results, PD^2 and S-PD^2 perform poorly in comparison to the EDF-based algorithms. However, with the smaller WSS (inset (d)), the Pfair algorithms exhibit reasonable performance. Third, even with light tasks, which is the easy case for P-EDF, when total utilization is high, G-EDF and NG-EDF are able to schedule significantly more task sets than P-EDF. For instance, when the total base utilization is 3.8, in inset (a), the schedulability curves for G-EDF and NG-EDF are higher than that of P-EDF by roughly 60% and 90%, respectively. Fourth, referring to insets (b) and (e), when *all* tasks are heavy, schedulability drops to close to 0% for P-EDF. This is due to the fact that, in most cases, at most one task fits on each processor, and hence, a task system with more than four tasks cannot be partitioned among four processors. The other four algorithms exhibit reasonable performance in these insets, with the EDF schemes performing better than the Pfair schemes, due to the higher preemption and migration costs of the latter. Fifth, the Pfair schemes perform better when task utilizations are higher. For instance, in inset (b), when the total utilization is in [3.0,3.4], PD^2 and S-PD^2 are capable of scheduling close to 100% of all task sets in comparison to 0% in inset (a) with light tasks. Finally, when the task system is not exclusively heavy but contains some light tasks, P-EDF performs significantly better than when all tasks are heavy (insets (c) and (f)). However, its performance is still worse than that of each of the other four algorithms.

4.3 Backlogged Performance Experiments

Finally, we compared the raw performance of each scheme on LITMUS$^{\mathrm{RT}}$ when scheduling continuously-backlogged tasks that execute the simple program BACKTEST listed in Fig. 11 (and discussed below) on three different task sets. To concisely describe these task sets, we use the notation "$n_1 \times (e_1, p_1), n_2 \times (e_2, p_2), \ldots$" to denote that n_1 tasks are included with an execution cost of e_1 and period of p_1, and so on. The three sets are: **Set (A)**, with tasks $28 \times (10, 500)$, $12 \times (35, 350)$, $12 \times (12, 240)$, and $32 \times (9, 300)$; **Set (B)** with tasks $4 \times (76, 300)$, $4 \times (4, 16)$, $5 \times (9, 90)$, and $25 \times (10, 500)$; and **Set (C)** with tasks $4 \times (18, 30)$, $5 \times (4, 40)$, and $25 \times (4, 200)$. These task sets are all schedulable under each scheme considered in this paper. Set (A) is comprised solely of light tasks of weight at most 0.1, Set (B) includes "moderate" weights of up to 0.256, and Set (C) includes both light and heavy tasks. For each task set and scheduling algo-

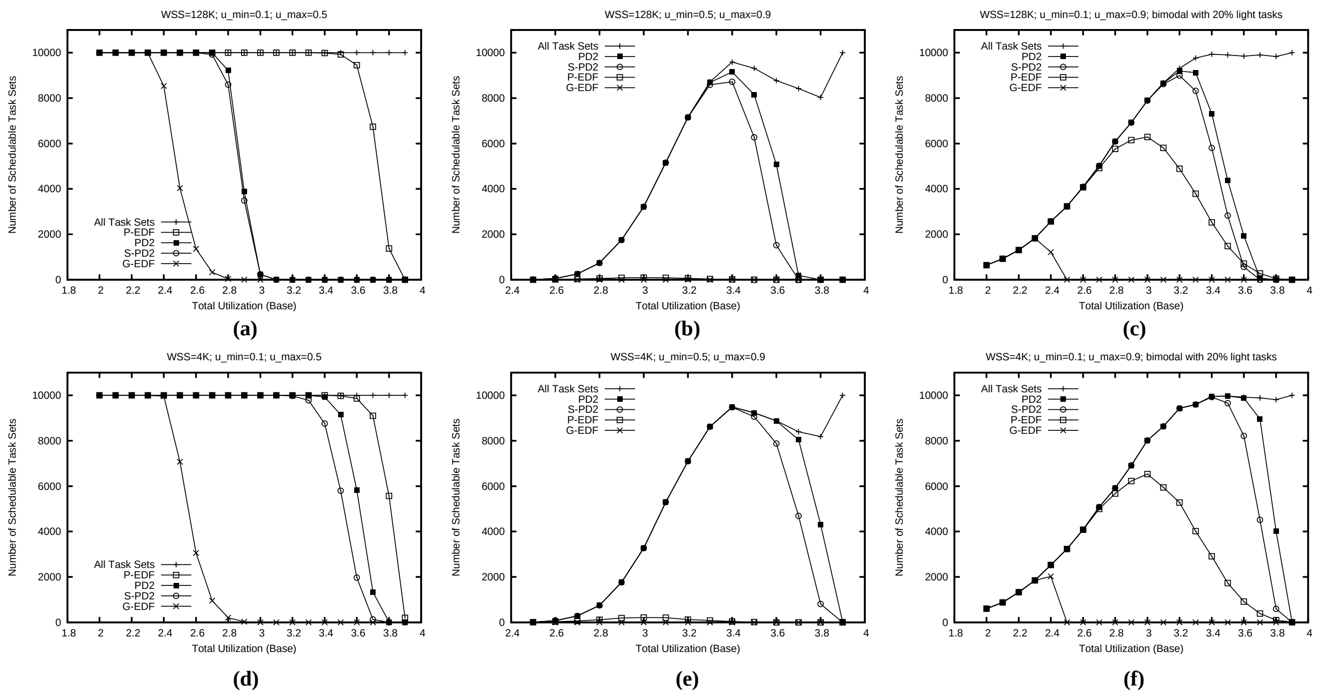


Figure 9: NST results for hard-real-time systems with WSS=128K in insets **(a)**, **(b)**, and **(c)** and WSS=4K in insets **(d)**, **(e)**, and **(f)**. Per-task utilizations are uniformly distributed in the range $[0.1, 0.5)$ in insets (a) and (d) and in the range $[0.5, 0.9)$ in insets (b) and (e), and are bimodally distributed between the ranges $[0.1, 0.5)$ and $[0.5, 0.9)$ with probabilities 0.2 and 0.8, respectively, in insets (c) and (f). In each inset, the order of the legend is the same as that of the curves. 99% confidence intervals were computed but have been omitted as their ranges are minimal and their inclusion obscures the identification marks of the different curves.

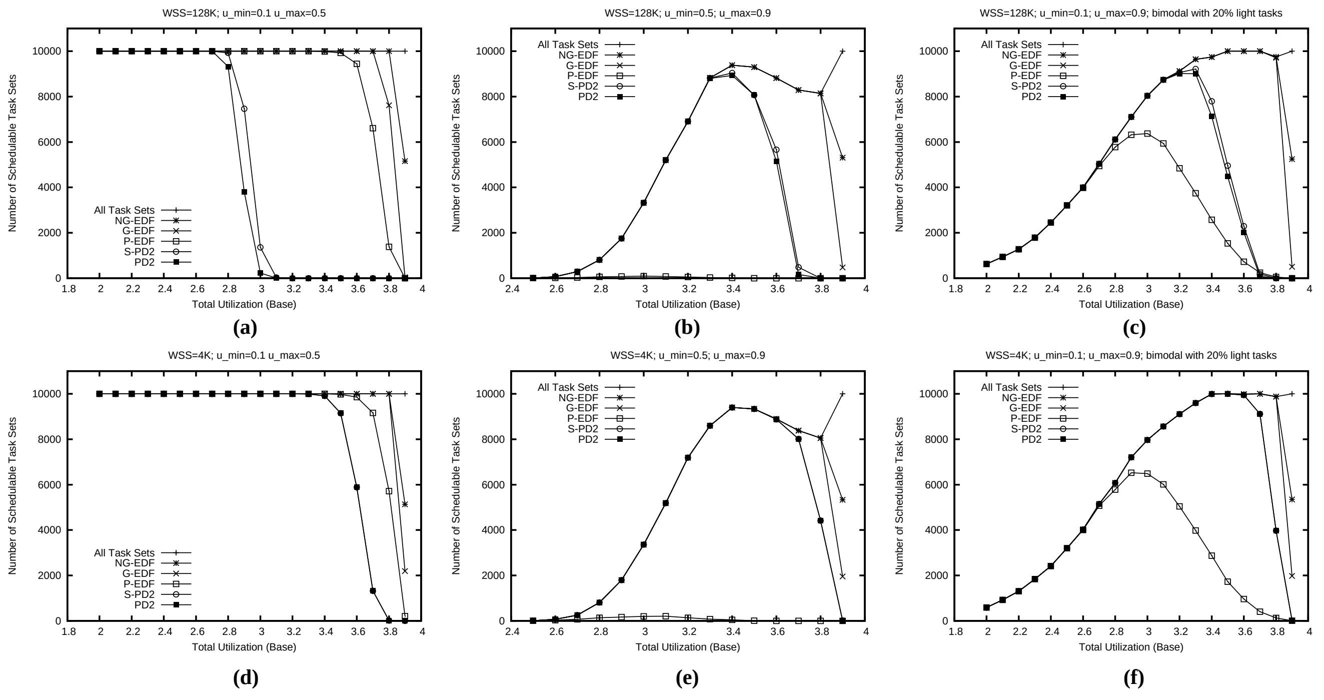


Figure 10: NST results for soft-real-time systems. The insets are organized in the same way as in Fig. 9.

```
BACKTEST ($iArray$: **array of integers**, $wt$: **rational**)
1  $i := 0$;
2  $T := 0$;
3  $size :=$ SIZEOF($iArray$);
4  **while** $i < 5{,}000{,}000 \cdot wt$ **do**
5      $M_1 :=$ RAND() $\mathbf{mod}$ $(size)$;
6      $M_2 := size - M_1 - 1$;
7      $T := T + (iArray[M_1] - iArray[M_2])$;
8      $i := i + 1$
   **od**
```

Figure 11: Test function. RAND generates random integers.

rithm, we recorded the completion time of every task, from which we computed the *average task completion time*, denoted ATC, and its standard deviation. A large ATC is indicative of poor performance, and a large standard deviation implies that task completion times varied significantly. To ensure that P-EDF is considered fairly, we partitioned tasks so that the total utilization on each processor is the same. This reduces the likelihood that one heavily-loaded processor will result in an increased ATC value.

BACKTEST randomly accesses elements in an integer array $iArray$, the size of which determines the array size (AS) of each task. BACKTEST accesses two distinct elements from $iArray$ during each loop iteration. As a result, up to two blocks of memory from each task's array are brought into cache on every iteration. Thus, a randomly-accessed element of $iArray$ has a greater probability of being in the cache, as the cache is "warmed-up" faster. Note that the number of iterations that each task must perform is scaled by its weight, so in an "ideal" system, all tasks would complete at the same time. (It would be desirable to consider other memory-access patterns, but space constraints prevent this.)

Recorded ATC values with standard deviations for three ASs are shown in Fig. 12. We note five important behaviors. First, perhaps somewhat unexpectedly, the ATC value for P-EDF is *not* the smallest for any AS. This is because, despite our attempts to treat P-EDF fairly, the tasks on one processor may take a long time to complete relative to the tasks on all other processors, thus increasing the overall completion time. This is also the reason for the large standard deviations recorded for P-EDF. Second, for Set (C), PD^2 has the lowest ATC of any scheme. This is because heavy tasks under PD^2 are likely to be scheduled in successive time slots, which reduces the number of preemptions and migrations. Third, for every task set and AS, NG-EDF has a higher ATC value than G-EDF. This is because the increased tardiness of NG-EDF cancels any gain due to lower migration and preemption costs. Fourth, for Set (A), S-PD^2 has the best ATC value of any scheme. This is because, as the number of tasks increases, the likelihood of a task being preempted under any scheme (except NG-EDF) increases. Since S-PD^2 incurs smaller costs per migration or preemption, its performance is impacted less by an increase in the number of migrations/preemptions. Finally, and perhaps surprisingly, the two Pfair schemes exhibit performance that compares reasonably to the others in terms of ATC, and both tend to perform better in terms of standard deviation (perhaps due to the "fairness" enforced by these algorithms).

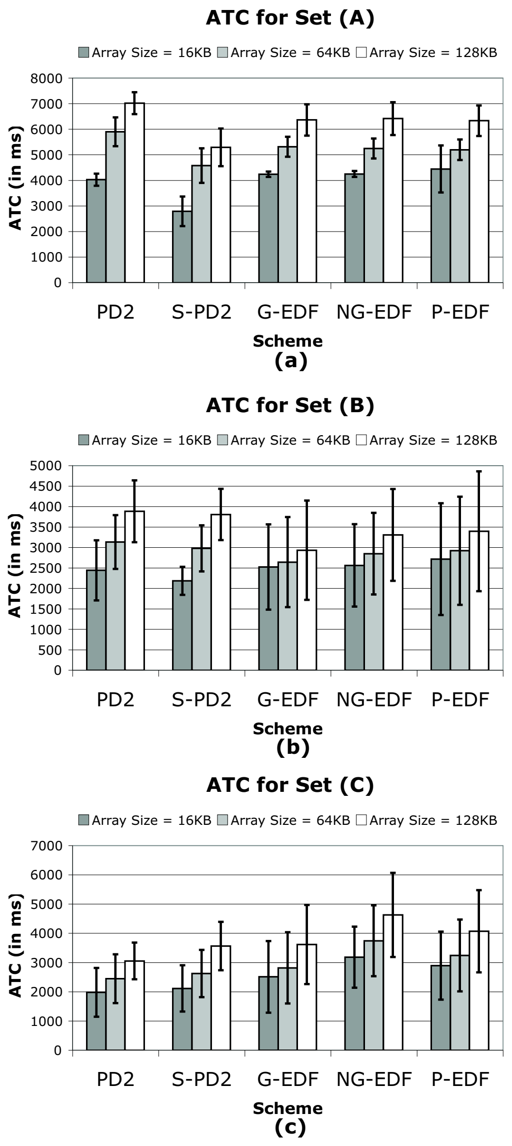


Figure 12: ATC values (in ms) and standard deviations for three task sets ((A), (B), and (C)) and three ASs (16 KB, 64 KB, and 128 KB).

5 Concluding Remarks

In this paper, we presented the LITMUS$^{\text{RT}}$ testbed, and discussed the results of experiments using LITMUS$^{\text{RT}}$ to compare a number of multiprocessor real-time scheduling algorithms. These experiments suggest that global algorithms are a viable alternative to partitioning approaches. In fact, our results clearly show that for each algorithm, scenarios exist in which it is the preferred choice.

There are numerous directions for future work. First, we would like to extend LITMUS$^{\rm RT}$ to include support for separate address spaces (if needed), task synchronization and communication, non-periodic workloads, and dynamic behavior (*e.g.*, task-set changes). (Regarding non-periodic workloads, our implementation of LITMUS$^{\rm RT}$ should enable event-driven sporadic tasks to be easily supported.) Second, we wish to improve the predictability of LITMUS$^{\rm RT}$ by further constraining interference due to interrupts. Third, we would like to evaluate other multiprocessor algorithms (*e.g.*, static-priority algorithms). Fourth, we want to re-assess the overheads considered in this paper on larger platforms and on other architectures, particularly multicore platforms, and determine if our experimental results apply to them as well. Fifth, we want to determine average case overheads for various classes of real-time applications and determine how using these overheads influences our results. Sixth, we would like to explore the viability of using tickless scheduling and one-shot timers in EDF schemes. Seventh, we would like to explore the potential of applying staggering to the non-Pfair schemes. This would likely not reduce average preemption and migration costs, but it might ease worst-case scenarios that would have to be considered when determining system overheads; of course, such gains would come at the expense of some losses, such as partially-used quanta. Eighth, we would like to experiment with a greater variety of workloads than those we have considered to date. Finally, we want to document LITMUS$^{\rm RT}$ and improve its API, with the goal of eventually producing an extension that could be posted online, *e.g.*, at SourceForge.

Regarding multicore platforms, it is worth noting that, on such platforms, some of the overheads considered in this paper may be less of a concern. For example, the main cost associated with a task migration is a loss of cache affinity. However, most existing and proposed multicore architectures include shared on-chip caches. The cost of accessing such a cache is just a few tens of cycles, in comparison to 100-300 cycles for off-chip memory. Thus, losing affinity with respect to dedicated per-core caches in these systems is less of an issue than in a traditional SMP. This further strengthens the case for global scheduling approaches. We expect that our future experimental efforts will validate this conclusion.

References

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of Linux. *Proc. of the Real-Time Technology and Applications Symp.*, 2002.

[2] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of asynchronous periodic tasks. *JCSS*, 68(1):157–204, 2004.

[3] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Department of Computer Science, Florida State University, 2005.

[4] T. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proc. of the 24th IEEE Real-time Systems Symp.*, pp. 120–129, 2003.

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

[6] A. Block, J. Anderson, and U. Devi. Task reweighting under global scheduling on multiprocessors. In *Proc. of the 18th Euromicro Conference on Real-Time Systems*, 2006.

[7] D. Bovet and M. Cesati. In *Understanding the Linux Kernel, 3rd edition*. O'Reilly Publishers, 2005.

[8] J. Calandrino and J. Anderson. Quantum support for multiprocessor Pfair scheduling in Linux. In *Proc. of the 2nd Int'l Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

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

[10] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proc. of the 26th IEEE Real-time Systems Symp.*, pp. 330–341, 2005.

[11] DIAPM, Dipartimento di Ingegneria Aerospaziale Politecnico di Milano. A hard real time support for Linux. 2002.

[12] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[13] P. Holman and J. Anderson. Adapting Pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.

[14] J. Lopez, M. Garcia, J. Diaz, and D. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proc. of the 12th Euromicro Conference on Real-time Systems*, pp. 25–33, 2000.

[15] S. Shankland and M. Kanellos. Intel to elaborate on new multicore processor. http://news.zdnet.co.uk/hardware/chips/0,39020354,39116043,00.htm, 2003.

[16] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. In *Proc. of the 34th ACM Symp. on Theory of Computing*, pp. 189–198, 2002.

[17] J. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3):237–253, 2004.

[18] J. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Computer*, 8(3):62–72, 1991.

[19] J. Stohr, A. von Bulow, and G. Farber. Using state of the art multiprocessor systems as real-time systems—the RECOMS software architecture. *Work-in-progress proc. of the 16th Euromicro Conference on Real-Time Systems*, 2004.

[20] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proc. of the 26th IEEE Real-time Systems Symp.*, pp. 311–320, 2005.

[21] V. Yodaiken and M. Barabanov. A real-time Linux. In *Proc. of the Linux Applications Development and Deployment Conference (USELINUX)*. The USENIX Association, 1997.