LITMUS$^{\mathrm{RT}}$: A Status Report*

Björn B. Brandenburg, Aaron D. Block, John M. Calandrino,
UmaMaheswari Devi, Hennadiy Leontyev, and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

This paper describes a real-time extension to Linux called LITMUS$^{\mathrm{RT}}$, which is being designed to support real-time workloads on multiprocessor and multicore platforms. The recent shift by chip makers to multicore designs, combined with building interest within the open-source community in supporting real-time features in Linux, makes this research quite timely. The development of LITMUS$^{\mathrm{RT}}$ was driven by a desire to bridge the gap between those working on algorithmic issues pertaining to multiprocessor real-time resource allocation and operating-systems researchers working to improve real-time support within operating systems such as Linux.

1 Introduction

In this paper, we report on the development of a real-time extension of Linux called LITMUS$^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems) [9, 11, 28], which is being designed to support real-time workloads on multiprocessor platforms. The development of LITMUS$^{\mathrm{RT}}$ has been driven by several trends. Foremost among these is the advent of multicore platforms as an alternative to single-core chip designs. Most (if not all) major chip manufacturers have embraced multicore technologies as a way to continue performance improvements in their product lines. Given this trend, multiprocessors will soon become the "standard" computing platform in many settings, including settings where real-time constraints are required. Indeed, IBM's multicore Cell processor was originally designed for gaming systems, where timing constraints naturally arise. In more general-purpose settings, one envisioned use of multicore platforms is as *multi-purpose home appliances*, with one machine serving many of the computing needs within a home [10]. These may include time-sensitive and computationally-intensive computations such as HDTV-quality media streaming, in addition to non-real-time processing.

Another relevant trend is the surge of interest in the open-source community in real-time variants of Linux. In conjunction with this, a number of real-time extensions of Linux have been proposed [22, 30]. In addition, features such as high-resolution timers, priority inheritance, and shortened non-preemptable sections, which enhance real-time predictability, have been incorporated in the Linux kernel (in versions 2.6.16 to 2.6.22). Further improvements in supporting real-time execution are likely, as there is now a sizable community of researchers interested in implementing new real-time-oriented features in Linux (as evidenced by the existence of the workshop in which this paper appears).

Many of the real-time Linux variants under development will be deployable on multicore and multiprocessor platforms. Unfortunately, most of these variants have been produced without much regard to recent algorithmic advances in work on multiprocessor real-time resource allocation. For example, global scheduling policies (which schedule tasks from a single run queue) are almost never implemented, despite the fact that such policies are *provably* superior to partitioning approaches in many ways (as we discuss later). This "mis-match" between theory and practice cannot be blamed solely on experimentalists. Indeed, in the last few years, scores of papers have been written on multiprocessor real-time scheduling algorithms (too many for us to include a citation list), yet working implementations do not exist for many (if not most) of the algorithms that have been recently proposed.

The development of LITMUS$^{\mathrm{RT}}$ has been mostly driven by a desire to bridge this gap between operating-systems researchers and those working on algorithmic issues. In addition, our research group is actively investigating real-time resource-allocation issues of relevance to multicore platforms and we seek to use LITMUS$^{\mathrm{RT}}$ as a test platform in this work. LITMUS$^{\mathrm{RT}}$ is an extension of Linux (currently, version 2.6.20) that allows different (multiprocessor) scheduling algorithms to be linked as plug-in components. In addition, a new multiprocessor real-time locking protocol of our own design has also been implemented in LITMUS$^{\mathrm{RT}}$. Although the current LITMUS$^{\mathrm{RT}}$ version is very much a prototype, our ultimate goal is to extend it in ways that result in a feature-rich system that is capable of supporting complex real-time applications on multicore platforms.

In prior work, our research group has used the cur-

*Work supported by Intel Corp., NSF grants CNS 0408996, CCF 0541056, and CNS 0615197, and ARO grant W911NF-06-1-0425.

rent LITMUS$^{\mathrm{RT}}$ system as a test platform [4, 9, 11, 28]. However, in these prior papers, we have not had sufficient space to describe the current LITMUS$^{\mathrm{RT}}$ implementation or the rationale behind its design in any detail. Such is the objective of this paper. Towards this end, the rest of this paper proceeds as follows. We begin with an overview of relevant real-time systems concepts related to scheduling and synchronization in Section 2. Next, in Section 3, we describe our overall objectives in producing LITMUS$^{\mathrm{RT}}$. Then, in Section 4, we explain in some detail the various implementation choices that underlie the current LITMUS$^{\mathrm{RT}}$ design. We conclude the paper and discuss our plans for improving the current LITMUS$^{\mathrm{RT}}$ implementation in Section 5.

2 Background

In this section, we provide background on real-time systems that is needed to understand our implementation.

2.1 Real-Time Systems Basics

For the most part, we focus in this paper on the problem of supporting a real-time workload on m processors that can be specified as a collection of *sporadic tasks*, denoted $T_1, \ldots, T_N$. (We note, however, that LITMUS$^{\mathrm{RT}}$ currently supports other task/job models as well. We provide brief explanations of these models later.) Each task in a sporadic system is invoked or *released* repeatedly; each such invocation is called a *job* of the task. Each sporadic task T_i is specified by a *period*, $\mathsf{p}(T_i)$, which denotes the minimum separation between its successive job releases, and by an *execution cost*, which denotes the maximum execution time of any of its jobs. The j^{th} job (or invocation) of task T_i is denoted T_i^j. T_i^j becomes available for execution at its *release time*, $\mathsf{r}(T_i^j)$, and should complete execution by its *absolute deadline*, $\mathsf{r}(T_i^j) + \mathsf{p}(T_i)$; otherwise, it is *tardy*. The spacing between job releases must satisfy $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$. If the stronger requirement $\mathsf{r}(T_i^{j+1}) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i)$ is always met, then the task system is called *periodic*. Task periods and job release times (even for sporadic tasks) 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. A task's utilization reflects the processor share that it requires. Task utilizations are of importance when checking schedulability, *i.e.*, whether timing constraints are met.

A *hard* real-time system is considered to be *schedulable* iff it can be shown that no job deadline is ever missed. A *soft* real-time system is considered (in this paper) to be *schedulable* iff it can be shown that deadline tardiness is bounded, that is, some value B exists such at all jobs are guaranteed to complete by B time units after their deadlines. Algorithms that are used to check schedulability must be designed to account for overheads that arise in practice. Sources of such overheads include context switching times, cache-related overheads, *etc.* Such overheads are typically accounted for by inflating per-job execution costs.

Real-time guarantees. In prior work on adding real-time support within Linux, much attention has been directed at increasing the predictability of certain components of Linux in such a way that the impact of system overheads (*e.g.*, interrupt latency) is limited or bounded. While reducing system latency is important and will ultimately improve the real-time guarantees that can be made, they are not of themselves enough to support real-time tasks—note that "real fast" is not the same as real-time. (In fact, for many real-time workloads, additional utility is not gained by improving response times beyond what is needed to ensure that all timing constraints are met.) Real-time tasks need explicit support from the scheduler so that guarantees related to their timing constraints can be made. The LITMUS$^{\mathrm{RT}}$ project implements a variety of scheduling algorithms that allow us to make such guarantees.

Overview of multiprocessor scheduling. Multiprocessor real-time scheduling algorithms can be divided into two categories: those that *partition* the task set, statically assigning tasks to processors, and *global* approaches that schedule tasks from a single run queue and allow migration. Several multiprocessor scheduling algorithms have been implemented in LITMUS$^{\mathrm{RT}}$. Most of these are based on the uniprocessor earliest-deadline-first (EDF) scheduling algorithm, in which jobs with earlier deadlines have higher priority. These include: partitioned EDF (P-EDF), and preemptive and non-preemptive global EDF (G-EDF and G-NP-EDF). (Two variants of P-EDF and G-EDF, called PSN-EDF and GSN-EDF, respectively, have been implemented as well. We briefly explain the need for these variants later.) In addition, two variants of the global PD^2 Pfair algorithm [1] have been implemented.

In P-EDF, tasks are statically assigned to processors and those on each processor are scheduled on an EDF basis. In G-NP-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 G-NP-EDF and G-EDF is bounded in such systems [14, 29].

In contrast to EDF-based algorithms, optimal scheduling is possible with Pfair scheduling algorithms [3, 27], *i.e.*, deadline misses can be completely avoided in any feasible system. In such algorithms, 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 over which it is continuously active (*i.e.*, submitting jobs). 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 [1, 27], which uses two tie-breaking rules. Two variants of PD^2 are implemented in LITMUS^{RT}: *synchronized* PD^2 (which we simply denote as PD^2) and *staggered* PD^2 (denoted S-PD^2) [19]. 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 when 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 that all subtask deadlines for any feasible system are met, 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 G-NP-EDF [14, 29]. 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, G-NP-EDF, and S-PD^2, a deadline is missed. Third, in the G-NP-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.

Partitioning versus global scheduling. Global scheduling algorithms are better able to utilize multiprocessor systems than partitioning approaches when system overheads are negligible. For example, as noted earlier, PD^2 can schedule on m processors any sporadic task system with total utilization at most m [1], and G-EDF and G-NP-EDF can ensure bounded deadline tardiness for any such task system, again, if total utilization is at most m [15]. 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 [15].

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 developing LITMUS^{RT} has been to determine whether this viewpoint is warranted. In particular, we wanted to know how partitioning and global real-time scheduling approaches compare when real overheads, empirically determined, are considered.

In [11], we report on results obtained using LITMUS^{RT} on a four-processor testbed to compare the five multiprocessor scheduling algorithms described above. 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 pre-

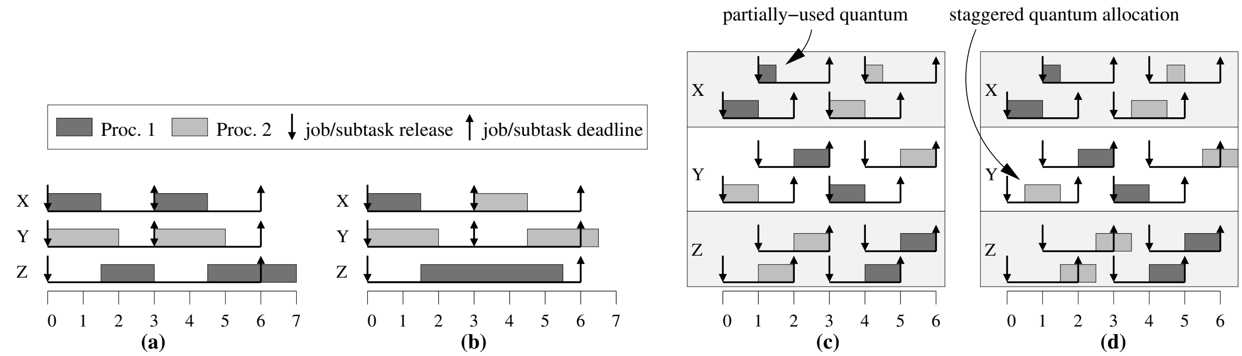

Figure 1: **(a)** G-EDF, **(b)** G-NP-EDF, **(c)** PD^2, and **(d)** S-PD^2 schedules of a two-processor system of three tasks: X, with an execution cost of 1.5 and period of 3.0, Y with an execution cost of 2.0 and a period of 3.0, and Z with an execution cost of 4.0 and a period of 6.0.

dictability. We found that, for hard real-time systems, P-EDF and PD^2 are usually preferable, while for soft real-time systems, G-EDF and G-NP-EDF are better. In the hard real-time case, most partitioning and non-Pfair global algorithms have rather similar schedulability tests in the absence of overheads (a survey of such tests can be found in [12]). As a result, partitioning approaches tend to be preferable because they have lower run-time overheads [11]. In addition, the optimality of PD^2 tends to compensate for its higher runtime overheads. In contrast, in the soft real-time case, P-EDF is subject to bin-packing limitations, to which G-EDF and G-NP-EDF are immune. In addition, G-EDF and G-NP-EDF benefit in comparison to PD^2 because they have lower runtime overheads.

2.2 Real-Time Synchronization

We now consider the issue of how to synchronize accesses to shared resources in multiprocessor real-time systems. Of the available options for doing this, locking mechanisms are clearly the most commonly used. In recent work, members of our research group devised a new multiprocessor real-time locking scheme called the *flexible multiprocessor locking protocol* (FMLP) [5]. The FMLP has been implemented in LITMUS$^{\mathrm{RT}}$, and an empirical comparison of it to non-blocking approaches has been conducted as well [9]. We discuss these research efforts in some detail below, after first providing needed background.

Resources and shared objects. When locks are used, jobs *issue requests* for exclusive access to resources. If a request is not satisfied immediately, then the issuing job is *blocked*. Once satisfied, the issuing job *holds* the resource until it completes its associated *critical section* and *releases* the resource. A request R is *contained* (or *nested*) within another request R' if the requesting job already holds R' when it requests R. A request is *outermost* if it is contained within no other request.

In lock-based synchronization schemes, blocking, by spinning or suspension, is inherent. Spin-based locking algorithms are commonly called *spin locks*. Of greatest interest here are FIFO spin locks known as *queue locks*, wherein blocked tasks wait within a FIFO queue of spinning tasks [24]. Such locking algorithms are designed so that all spinning is *local*, *i.e.*, via read-only spin loops that (in the absence of preemption) give rise to only a constant number of shared-memory accesses when used in systems with coherent caches or distributed shared memory. Spin locks can be used by tasks with little (or no) interaction with the kernel. In contrast, suspension-based blocking is used in OS-based synchronization protocols in which resources are acquired and released via system calls.

The literature on lock-based synchronization is vast and includes (for example) mechanisms that are hybrids of pure spin-based and suspension-based mechanisms (*e.g.*, [20]). However, for our purposes, a locking mechanism must have *analyzable* blocking behavior so that job blocking times can be accounted for when checking schedulability. Thus, mechanisms derived in work on non-real-time systems for which the required analysis does not exist are of no interest to us.

Prior synchronization-related work. Rajkumar *et al.* [25] were the first to propose locking protocols for real-time multiprocessor systems. They presented two multiprocessor variants of the priority-ceiling protocol (PCP) [26] for systems where partitioned, static-priority scheduling is used. In later work, several protocols were presented for systems scheduled by P-EDF. The first such protocol was presented by Chen and Tripathi [13], but it is limited to periodic (not sporadic) task systems. In later work, Lopez *et al.* [21] and Gai *et al.* [17] presented protocols that remove such limitations, at the expense of imposing certain restrictions on critical sections (such as, in [17], requiring all global critical sections to be non-nested). A scheme for G-EDF that is also restricted was presented by Devi *et al.* [16]. The FMLP, mentioned earlier, does not restrict the kinds of critical sections that can be supported and can be used under either G-EDF or P-EDF. In the FMLP, resources are protected by either spin-based or suspension-based locks. The FMLP is the only scheme known to us that is capable of supporting arbitrary critical sections under G-EDF. Furthermore, the schemes in [16, 17, 21] are special cases of it.

The FMLP. We now provide an overview of the FMLP. It is not our intent here to describe every detail of this protocol—a full description of it can be found in [5]. Instead of repeating that description here, we instead have opted to explain how the design choices underlying the FMLP were made. Such a description should (hopefully) suffice when trying to understand the description of our implementation of synchronization in LITMUS$^{\mathrm{RT}}$, given later.

The FMLP is considered to be "flexible" for two reasons: it can be used under either partitioned or global scheduling, and it is agnostic regarding whether blocking is via spinning or suspension. Regarding the latter, resources are categorized as either "short" or "long." Short resources are accessed using queue locks and long resources are accessed via a semaphore protocol. Whether a resource should be considered short or long is user-defined, but requests for long resources may not be contained within requests for short resources. The terms "short" and "long" arise because (intuitively) spinning is appropriate only for short critical sections, since spinning wastes processor time. However, the experimental results presented in [9] call this view into question.

The remaining details underlying the design of the FMLP were resolved with the express purpose of trying to ease the task of calculating worst-case job blocking times. In this regard, *simple mechanisms* are much more desirable than complex ones: with complex mechanisms, very conservative assumptions must be made when determining blocking times, and as a result, estimated blocking times may grossly overestimate those observed in practice. These estimates are very important because they determine the impact of synchronization when checking system schedulability.

With this in mind, the FMLP was designed by systematically considering a number of issues, and for each, considering different design choices. In each case, the choice that was adopted was that which resulted in better blocking-time estimates. From these design decisions, a number of underlying principles of the FMLP emerged, as listed below.

- *Discourage preemptions of resource-holding jobs.* When a resource-holding job is preempted, other jobs waiting for the same resource may be substantially delayed. Thus, in the FMLP, such preemptions are discouraged. With one exception, this is done by actually executing resource requests *non-preemptively*. The exception is long resources under G-EDF, for which *priority inheritance* is used instead: a job that holds a resource inherits the priority of the highest-priority job it blocks. Priority inheritance is not used under P-EDF because priorities on different processors cannot be meaningfully compared (two jobs on different processors with equal deadlines may have very different priorities from a per-processor perspective: one may have the highest priority on its processor, and the other the lowest priority on its processor). Note that, in the case of long resources under P-EDF, a requesting job executes non-preemptively only if it is not suspended. (Suspensions are not an issue for short resources.) The group-locking mechanism discussed below ensures that such a job suspends at most once per outermost request.

- *Prioritize lock requests on a FIFO basis.* If lock requests are ordered on an EDF basis, then it can be difficult to bound blocking times. In particular, a job's blocking time would depend on future higher-priority job arrivals; usually conservative assumptions are made regarding such arrivals, which can result in high blocking-time estimates. The FMLP instead prioritizes requests in FIFO order. With FIFO ordering (and non-preemptive execution) on m processors, a request can be blocked by at most $m-1$ preceding requests. In most systems, m will be rather small, and hence this bound is quite close to being tight.

- *Use a* (*very*) *simple deadlock-avoidance mechanism.* It can be difficult to accurately bound blocking times when complex deadlock-avoidance mechanisms are used (such as priority-ceiling-related mechanisms [25]). In the FMLP, deadlock is prevented by "grouping" resources and allowing only one job to access resources in any given group at any time. Two resources are in the same group iff they are of the same type (short or long) and requests for one may be nested within those of the other. A *group lock* is associated with each resource group; before a job can access a resource, it must first acquire its corresponding group lock.

For short resources, group locks are acquired using queue locks, and for long resources, they are acquired using a semaphore protocol. Note that, in the case of nested resource requests, all blocking incurred by a job occurs when it attempts to acquire the corresponding group lock.

Under P-EDF, it is possible that all tasks that request long resources from a given group may be assigned to the same processor. Such long resources are called *local* (others are called *global*). In dealing with local resources under P-EDF, Baker's uniprocessor stack resource protocol (SRP) [2] is used in the FMLP instead of the more complex mechanisms outlined above. Lopez *et al.* [21] were the first to propose this optimization. Note that, since there is no notion of locality under G-EDF, this technique cannot be used under it. It is worthwhile to note that under P-EDF the synchronization protocol of Gai *et al.* [17] is equivalent to the FMLP when all long resource requests are local, and that of Lopez *et al.* [21] is equivalent to the FMLP when all long resource requests are local and there are no short resource requests.

Experimental evaluation. In recent work [9], we presented an empirical comparison of the long- and short-resource variants of the FMLP and lock-free and wait-free algorithms on our LITMUS$^{\rm RT}$ testbed. *Lock-free* and *wait-free* algorithms are user-level synchronization alternatives to locking that can be used when the resource in question is a shared data object. In lock-free and wait-free object implementations, object accesses may occur concurrently. In the lock-free case, such accesses may "interfere" with each other, and accesses that experience interference must be retried. In the wait-free case, object accesses are implemented in a non-blocking way that ensures that each access completes in a bounded number of steps (statement executions). In the evaluation in [9], we first obtained system and synchronization overheads by running benchmarks on LITMUS$^{\rm RT}$. Using these overheads, we then conducted two sets of schedulability experiments. In each, both hard and soft real-time schedulability were considered.

In the first set of experiments, we considered only locking mechanisms. Our goal was to determine when (if ever) suspending is better than spinning. We considered a wide spectrum of lock nesting levels and critical-section durations. In these experiments, suspension-based locking *never* resulted in better schedulability than spin-based locking. (On the other hand, more processor time may be available to background jobs if suspension-based locking is used.) In the second set of experiments, we considered specifically the problem of implementing shared data objects. Our main objective here was to determine when (if ever) lock-free and wait-free techniques are preferable to locking techniques. Our study focused on three representative objects: read/write buffers, queues, and binary heaps (listed in order of increasing complexity). In this study, schedulability was generally better with locking, but wait-free implementations tended to be comparable (even for more complex objects for which wait-free implementations are often dismissed as impractical) and were even superior for simple objects (buffers). On the other hand, lock-free implementations were viable *only* for simple objects.

2.3 Real-Time Linux

As noted earlier, there has been much recent interest in real-time variants of Linux. In fact, too many approaches have been developed for us to be able to adequately discuss them all here. Further, there does not even appear to be a strong consensus on what constitutes a proper "real-time Linux." In practice, real-time products use various approaches ranging from using an unmodified stock kernel, nested OS architectures, where Linux is scheduled as the idle tasks of a real-time OS (RTOS), to intricate processor-allocation schemes where the underlying hardware is partitioned among real-time and non-real-time applications, potentially even among multiple OSs [22].

In this paper, by "real-time Linux," we mean modified versions of the stock Linux kernel with improved real-time capabilities that are the single top-level resource manager, not paravirtualized variants such as RTLinux[30] or L^{4}Linux[18], where real-time tasks are not actually Linux tasks, nor other architectures where actual real-time guarantees are not based on Linux itself. Stronger notions of "real-time" can be provided in such systems, at the expense of a more restricted and less familiar development environment.

Limitations of real-time Linux. To satisfy the strict definition of hard real-time, all worst-case overheads must be known in advance and accounted for. Unfortunately, this is currently not possible in Linux, and it is highly unlikely that it will ever be. This is due to the many sources of unpredictability within Linux (such as interrupt handlers and priority inversions within the kernel), as well as the lack of determinism on the hardware platforms on which Linux typically runs. The latter is especially a concern, regardless of the OS, on multiprocessor platforms. Indeed, research on timing analysis has not matured to the point of being able to analyze complex interactions between tasks due to atomic operations, bus locking, and bus and cache contention. Despite these observations, there are now many advocates of using Linux to support applications that require some notion of real-time execution. As noted by McKenney [23],

> *I believe that Linux is ready to handle applications requiring sub-millisecond process-scheduling and interrupt latencies with 99.99+ percent probabilities of success. No, that does not cover every imaginable real-time application, but it does cover a very large and important subset.*

3 LITMUS$^{\mathrm{RT}}$

In this section, we present an overview of LITMUS$^{\mathrm{RT}}$ and its design.

3.1 What is LITMUS$^{\mathrm{RT}}$?

LITMUS$^{\mathrm{RT}}$ is an extension of Linux that supports a variety of real-time multiprocessor scheduling policies. In its current state, it is most useful as a testbed within which different scheduling policies can be implemented and empirically evaluated—it is not yet a stable, production-ready system. However, our ultimate goal for LITMUS$^{\mathrm{RT}}$ is to create a stable system that supports complex real-time applications on multicore platforms.

LITMUS$^{\mathrm{RT}}$ is designed in such a way that adding support for additional scheduling policies is straightforward—indeed, some of our currently-supported scheduling policies were implemented and tested in well under a week. Thus far, LITMUS$^{\mathrm{RT}}$ has been used by our group to conduct the two empirical studies mentioned earlier in Section 2 [9, 11], and in two other efforts that involved implementing scheduling policies for workloads that cannot be specified using a simple sporadic task model [4, 8]. Overall, LITMUS$^{\mathrm{RT}}$ has proven to be very useful in our work as a highly-extensible real-time scheduling testbed, and we believe that it may also be useful to other researchers.

LITMUS$^{\mathrm{RT}}$ was implemented by modifying the Linux 2.6.20 kernel configured to run on a symmetric multiprocessor (SMP) architecture. 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.

Why provide real-time support in Linux? We chose to create our testbed by modifying Linux instead of an existing 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 LITMUS$^{\mathrm{RT}}$ as it evolves will include many real-time graphics and multimedia applications developed within our own department. The developers of those applications actually prefer Linux as a development platform.

Our objectives in designing LITMUS$^{\mathrm{RT}}$ are in agreement with the earlier-noted sentiments expressed by McKenney. Thus, while we purposely limit attention to deploying scheduling and synchronization algorithms for which formal analysis exists—such algorithms should *not* be the weakest link from the standpoint of timing correctness—we acknowledge that producing system designs in any Linux-based system in which real-time correctness is *guaranteed* with certainty is not feasible. Related to this, we expect systems to be provisioned in LITMUS$^{\mathrm{RT}}$ using experimentally-determined worst-case (average-case) values for execution costs and system overheads in the hard (soft) real-time case, instead of using analytically-determined, verified values. This is, in fact, the approach we have taken in our prior work. Thus, in LITMUS$^{\mathrm{RT}}$, the term "hard real-time" should really be interpreted to mean that deadlines are *almost never* missed, and "soft real-time" to mean that deadline tardiness *almost always* remains within some bound, even if individual tasks misbehave. These are stronger guarantees than provided by most real-time Linux variants in commercial use today.

3.2 Challenges

We next describe the challenges we faced in creating LITMUS$^{\mathrm{RT}}$.

Supporting the sporadic task model. One common requirement for all of our scheduling policies is a need to support the sporadic task model. In order to correctly support this model within LITMUS$^{\mathrm{RT}}$, job releases need to occur at times when they can be handled immediately. We implemented a tick-based scheduler with the tick representing both a scheduling quantum and the time between consecutive local timer interrupts. We require jobs to be released only at quantum boundaries, so that they can be handled during local timer interrupts. To achieve this, jobs need to have periods that are an integral number of scheduling quanta; however, since scheduling decisions can be made between quantum boundaries when a job completes, execution costs are allowed to be non-integral.

Aligned quanta. Some multiprocessor real-time scheduling algorithms such as PD^2 require *synchro-*

nized quanta support, *i.e.*, their correctness relies on the assumption that different processors experience timer interrupts at the same time. Other algorithms that do not necessarily require such support may benefit from the existence of a uniform time base across processors so that job releases are observed by all processors at the same time. (One exception to this rule is S-PD^2, which requires staggered quanta.) Standard Linux does not provide aligned quanta—in fact, aligned quanta are not desirable in a purely throughput-oriented system due to bus-contention issues.

I/O support. Research on multiprocessor real-time scheduling analysis has yet to produce effective ways for accounting for disruptions caused by I/O. Nonetheless, support for I/O is crucial for real implementations. This is especially important in our case since the primary use of our test platform is for research, where logging data to stable storage is a necessity in many cases.

Deterministic synchronization. Spin-based locking in LITMUS$^{\mathrm{RT}}$ (as provided by the FMLP) is implemented using queue locks. Unfortunately, to handle most internal synchronization, the Linux kernel uses non-FIFO spin locks. This adds a source of non-determinism that could be substantial in some cases. Currently, it does not appear feasible to replace all spin locks inside the kernel with queue locks. Thus, we must be aware of their potential impact on the real-time guarantees that can be made.

3.3 The Design of LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ has been implemented via changes to the Linux kernel and the creation of user-space libraries. Since LITMUS$^{\mathrm{RT}}$ is concerned with real-time scheduling, most kernel changes affect the scheduler and timer interrupt code. The kernel modifications can be split into roughly three components. The *core infrastructure* consists of modifications to the Linux scheduler, as well as support structures and services such as tracing and sorted run queues that can be used by scheduler plugins. The *scheduler plugins* encapsulate the available real-time scheduling algorithms by providing functions that implement the methods of the scheduler plugin interface. Finally, a collection of *system calls* provides a user-space API for real-time tasks to interact with the kernel. In the following subsections, we describe each component in turn.

Note that, in the discussion that follows, the term *real-time task* means tasks that are scheduled by LITMUS$^{\mathrm{RT}}$. Normal Linux tasks that run with a static priority from the "POSIX real-time range" are not considered to be real-time tasks in LITMUS$^{\mathrm{RT}}$. Since they do not follow the sporadic task model, they are considered to be just best-effort tasks with a high static priority.

3.4 Core Infrastructure

Unlike with conventional OS scheduling algorithms, tasks are not always eligible to execute when scheduled with real-time algorithms. For example, a sporadic task that has completed a job may not be scheduled until its next job release. To facilitate the releasing and queuing of real-time tasks, LITMUS$^{\mathrm{RT}}$ provides the abstraction of a *real-time domain*, which is implemented by the abstract data structure `rt_domain_t`. `rt_domain_t` consists of a ready queue and a release queue (as well as one lock per queue). When a real-time domain is instantiated, it is parametrized with an *order function* that is used to sort tasks in the ready queue (the release queue is ordered by ascending release time). Most scheduling plugins in LITMUS$^{\mathrm{RT}}$ use the EDF order function. However, list sorting with various orders is used heavily in the *feedback-control EDF* (FC-EDF) algorithm, which is an adaptive scheduling algorithm briefly described later that was recently added to LITMUS$^{\mathrm{RT}}$ [4]. Wrapper functions are provided in the real-time domain for operations such as queuing, dequeuing, and inspecting designated queue elements. This removes the need for list-handling in most scheduler plugins, thereby reducing development effort (and also removing a common source of bugs).

To realize sorted (run) queues, LITMUS$^{\mathrm{RT}}$ extends the Linux `list.h` API with (parametrized) functions to insert an element into a sorted list (`list_insert()`) and to sort lists (`list_qsort()`).

Scheduling quanta are defined to be the intervals between local timer interrupts. To realize aligned quanta, LITMUS$^{\mathrm{RT}}$ synchronizes timer interrupts during boot across all processors. As explained in greater detail later, this is done by having each processor disable its local timer within the local timer interrupt handler, enter a barrier, and restart its timer immediately afterward. When all processors reach the barrier, they will be simultaneously released, resulting in all processors restarting their timers at approximately the same time. Using this method, we have been able to achieve aligned quanta with an error of at most 10 μs on our test platform—in some cases, error is as low as 1-2 μs. A slight alteration of this method can be used to realize staggered quanta, as required by S-PD^2.

Since LITMUS$^{\mathrm{RT}}$ is mainly intended as a research platform, strong introspection support is required to understand system behaviors. Thus, the core infrastructure provides several tracing facilities. With the $O(1)$-scheduler [6], `printk()` cannot be used while a run-queue lock is held. This limitation exists because `printk()` may invoke `try_to_wake_up()`, which will acquire run-queue locks to unblock the `syslogd` process. Unfortunately, most of the scheduling code executes while holding a run-queue lock, which makes debugging difficult. Accordingly, a macro infrastructure called `TRACE()` is provided in LITMUS$^{\mathrm{RT}}$ as an alternative to `printk()`-based debugging. The collected debug messages are exported to user-space via a custom character device driver. However, to avoid the recursive locking issues that plague `printk()`, polling is employed.

To obtain detailed insight into the schedules created by plugins, as well as to enable performance studies,[1] a framework called `sched_trace()` is provided to export a per-processor stream of scheduling events to user-space (also via a custom character device driver and realized with polling).

Finally, to record fine-grained overhead measurements, LITMUS$^{\mathrm{RT}}$ also contains a version of *Feather-Trace* [7], a static, lightweight tracing facility developed at UNC. While `TRACE()`- and `sched_trace()`-based logging can be disabled at compile time via configuration options, Feather-Trace is unintrusive enough to stay enabled at all times.

The core LITMUS$^{\mathrm{RT}}$ infrastructure also includes an implementation of the MCS queue lock [24]. Ideally, deterministic locking primitives should be used throughout the kernel, but this is problematic, as discussed earlier.

As the heart of LITMUS$^{\mathrm{RT}}$, the core infrastructure is also responsible for interfacing with the rest of Linux. It initializes a real-time scheduler plugin (based on a kernel command-line parameter) during system boot. To pass control to the plugin, it hooks into the Linux `scheduler_tick()` and `schedule()` functions. Overriding the Linux scheduler works as follows. Real-time tasks are assigned the highest static Linux scheduling priority upon creation. However, they are not kept in the standard Linux run queues. Instead each plugin is responsible for managing its own run queue. (Similarly, time-slice management is also delegated to plugins for real-time tasks.) When `schedule()` is invoked, control is passed to the current scheduler plugin. If it selects a real-time task to be scheduled on the local processor, then the task is inserted into the run queue and the Linux scheduler is bypassed. When a real-time task is preempted, it is removed again from the run queue, thereby taking it out of the reach of the Linux scheduler.

LITMUS$^{\mathrm{RT}}$ has two modes of operation, real-time and non-real-time. When started, the system is initially in non-real-time mode. Real-time tasks are not scheduled as long as the system is in non-real-time mode. This feature allows complete task systems to be set up before they are scheduled, thereby allowing for the synchronous release of the first jobs of all tasks. The core LITMUS$^{\mathrm{RT}}$ timer tick function (`rt_scheduler_tick()`) manages transitions to and from real-time mode.

3.5 Scheduler Plugins

As mentioned before, real-time scheduling policies are implemented as *scheduler plugins*. Such plugins are realized similarly to other pluggable components in Linux such as file systems. To create a scheduler plugin, functions that realize the thirteen methods[2] of the plugin interface defined by the struct `sched_plugin_t` and described below need to be implemented and registered by passing a pointer to an instance of `sched_plugin_t` to the LITMUS$^{\mathrm{RT}}$ core. Some of the methods are optional and do not need to be implemented by a plugin. From the plugins available, only one can be in use at any time. The choice is made at boot time based on the kernel command-line parameter `rtsched`. Switching to a different plugin at run-time, while possible in theory, is currently not implemented. The current version of LITMUS$^{\mathrm{RT}}$ contains the following scheduler plugins (the first four corresponding algorithms were described earlier):

1. P-EDF.
2. G-EDF.
3. G-NP-EDF.
4. PD^2 (and S-PD^2 when staggered quanta are enabled).
5. Partitioned EDF with synchronization support (PSN-EDF): similar to P-EDF, except that jobs cannot be preempted within critical sections, and the SRP is used to handle long local resources (see the earlier description of the FMLP).
6. Global EDF with synchronization support (GSN-EDF): similar to G-EDF, except that jobs cannot be preempted within critical sections, and priority-inheritance is employed for long resources. To bound the time that any job may be blocked due

[1] For example, we studied the impact of slack scheduling on average- and worst-case response times of best-effort jobs under EDF-HSB [8], which is an algorithm briefly described later that was designed for systems with both real-time and non-real-time components.

[2] Sometimes also called "operations" or "callbacks."

to non-preemptive sections in other jobs, the m highest-priority jobs on an m-processor system are *linked* to processors so that newly-arriving, high-priority jobs cannot preempt medium-priority jobs multiple times due to non-preemptive sections in low-priority jobs (see [5] for a detailed discussion of linking).

7. EDF for heterogeneous task systems (EDF-HSB): an EDF-based approach that uses P-EDF for hard-real-time tasks, G-EDF for soft-real-time tasks, and methods that minimize best-effort task response times.

8. Feedback-Control EDF (FC-EDF): an "adaptive" variant of G-EDF that uses feedback-control techniques to dynamically adjust task weights.

The thirteen methods that encompass the current scheduler-plugin interface are described below. As LITMUS$^{\mathrm{RT}}$ gains additional features, the number of methods will grow. For example, the addition of synchronization support introduced the last three methods listed below. By providing reasonable default implementations, existing plugins do not have to be changed when the interface is expanded.

1. When a new real-time task is added to the task set, the scheduler plugin is queried with a call to `prepare_task()`. This allows the plugin to examine the task and perform scheduler-specific initialization. If the task does not meet requirements imposed by the plugin, then it can veto the acceptance of the new task. This allows plugins to implement custom admission tests. However, plugins do not currently implement such tests.

2. Scheduler plugins are notified of tasks that block (for any reason) by calling `task_blocks()`. This allows plugins to remove blocking tasks from internal data structures. Most plugins do not need to act on this event because in Linux only the scheduled task can block, and scheduled tasks are usually not kept in the ready queue[3] in LITMUS$^{\mathrm{RT}}$.

3. The common Linux wake-up function `try_to_wake_up()` invokes the plugin method `wake_up_task()` if it determines that the task in question is a real-time task. LITMUS$^{\mathrm{RT}}$ makes sure that this method is only called once per blocking task, and only if `task_blocks()` was called previously, even if multiple calls to `try_to_wake_up()` are initiated. The mechanism works correctly even if the wake-up occurs before the task could block (*e.g.*, if the wake up occurs before the task in question could block by calling `schedule()` with its state set to `TASK_UNINTERRUPTIBLE`, which may happen when I/O operations complete very quickly).

4. When a real-time task exits, scheduler plugins are notified with a call to `tear_down`. To avoid memory leaks, a plugin should free any resources that it allocated for the exiting task.

5. When the system transitions to or from real-time mode, scheduler plugins are notified by a call to `mode_change()`. After being so notified, the plugin should place all real-time tasks in state `TASK_RUNNING` in the release queue for immediate release.

6. Real-time scheduler plugins are notified of each timer tick on each processor by a call to `scheduler_tick()`, regardless of whether a real-time task is scheduled or not. This enables scheduler plugins to release jobs and to preempt any task (real-time or non-real-time) at quantum boundaries.

7. The main scheduling function is `schedule()`. A plugin should select which real-time task to execute next. The Linux scheduler will only be consulted if this method selects no task.

8. To prevent race conditions, global schedulers must ensure that a task cannot be selected for execution on one processor before a context switch involving that task has finished on a different processor, otherwise stack corruption could occur. One way to ensure this is to only re-insert a preempted task into the ready queue after the context switch has completed. For that reason, `finish_switch()` is called after every context switch that involved a real-time task.

9. Individual jobs of a real-time task may complete early. In that case, the task should be put back into the release queue, where it must remain until its next job release. To notify a scheduler of such a condition, the method `sleep_next_period()` is invoked.

10. Certain schedulers, such as EDF-HSB, have parameters that affect how tasks are scheduled. In order to configure scheduler plugins through a unified interface, the method `scheduler_setup()`

[3] To be more accurate, there is one ready queue and one release queue under global algorithms, and one of each such queue per processor under partitioned algorithms.

was introduced. However, it is considered to be deprecated and will be removed in a future release because it duplicates functionality that is available through the `proc` file system.

The remaining three methods pertain to the implementation of the FMLP. Supporting priority-inheritance is optional for plugins, and currently only the plugins implementing the FMLP (PSN-EDF and GSN-EDF) implement this support.

11. When a task that was blocked on a long-resource group lock is unblocked, it may have to inherit the priority of a task that is located behind it in the FIFO wait queue for that group lock. To check for that condition, the long-resource code calls `inherit_priority()`.

12. When a task releases a long-resource group lock, it may have to relinquish an inherited priority. This is done by calling `return_priority()`.

13. When a task blocks on a long-resource group lock, it may have a higher priority than any other task in the wait queue for that group lock (if any). In that case, the priority of the lock-holder should be raised. The long-resource code calls `pi_block()` when a task blocks on a group lock to account for that situation.

3.6 System Call API

LITMUS^{RT} introduces a number of new system calls to Linux. While some of these system calls can be used directly, most of them are intended to be used by *liblitmus*, a user-space library that provides higher-level abstractions. The introduced system calls are organized by purpose into five groups: managing real-time tasks, querying state information, controlling job releases, system setup, and synchronization.

Real-time task management. Three system calls were introduced for real-time task management. Real-time tasks are created in three steps. First, a new task is created with the Linux `clone(2)` system call (the flag `CLONE_REALTIME` must be given). The child task will be started in the state `TASK_STOPPED` to give the creator time to properly configure the new real-time task. The parent task can then configure the child with either the `set_rt_task_param()` (for sporadic tasks) or the `set_service_levels()` (for adaptive tasks) system call. Once it is set up, the new task is added to the real-time task set with the system call `prepare_rt_task()`. Note that this API currently prohibits real-time tasks from configuring themselves. The reasons for this limitation are mostly historic, and we plan to make real-time task creation more flexible in a future release.

State information. Four system calls were added to allow real-time tasks to query information about the system and themselves. The currently-active scheduling policy can be obtained with `sched_getpolicy()`. Task-specific information can be obtained with `get_rt_task_param()`, which retrieves a sporadic task's parameters such as its worst-case execution time and period, `get_cur_service_level()`, which only applies to adaptive tasks, and `get_job_no()`, which returns the sequence number (starting at zero) for the task's current job.

Job control. There are two different system calls to signal the completion of a job. The simple one, `sleep_next_period()`, completes the current job unconditionally and places the invoking task on the release queue. This allows for a straightforward real-time task implementation. However, it has a subtle, potentially-unwanted behavior when a job overruns its allocation. In that case, the kernel will have already advanced to the next job by the time the job completion is signaled from user-space. Since `sleep_next_period()` works unconditionally, this will effectively skip the job after an overrun (which could be the desired behavior in some cases to control overload). To account for this situation, a second system call, `wait_for_job_no()`, was added that allows the job that should be released next to be specified. If that job has already been released (*e.g.*, due to an overrun of the previous job), then the system call returns immediately. It also allows a real-time task to skip several jobs by specifying a job release in the future.

System setup. Scheduler-specific settings can be configured with the `scheduler_setup()` system call. Mode transitions are initiated with the `set_rt_mode()` system call.

Synchronization. To support the FMLP, eleven system calls have been introduced. Eight of these provide support for long resources: `X_sema_init()`, for allocation, `X_sema_free()`), for de-allocation, `X_down()`, for acquisition, and `X_up()`, for release, where `X` is either `pi` (for non-SRP-controlled resources[4]) or `srp` (for SRP-controlled resources in

[4] `pi` stands for "priority inheritance." The `pi` system calls are also used for non-SRP-controlled resources in P-EDF, even though priority inheritance is not used for such resources.

P-EDF). Further, SRP-controlled resources also require real-time tasks to register their intent to access SRP-controlled resources so that priority ceilings can be correctly computed. This is done using the `reg_task_srp_sem()` system call.

To properly support spin-based resource access under the FMLP, real-time tasks need to become non-preemptive for short periods in user-space. We implemented non-preemptive sections by letting each real-time task register the address of a flag in user-space during initialization. This is done using the system call `register_np_flag()`. A task sets its flag prior to entering a non-preemptive section. When a delayed preemption is required because the task to preempt is executing non-preemptively (as indicated by its flag), the kernel sets a second flag in user-space. When a task leaves a non-preemptive section, it resets its flag and checks the kernel's flag. If it is set, then the task invokes the system call `exit_np()` to both reset the kernel flag and call the scheduler. This technique requires only one system call in the case of a delayed preemption, and zero otherwise.

4 Implementation

Our implementation efforts in developing LITMUS$^{\mathrm{RT}}$ have focused on several key tasks: devising support for different quanta alignments, incorporating multiprocessor scheduling algorithms into Linux using our plugin interface, providing support for various synchronization mechanisms, and developing user-space libraries that provide an interface for a user wishing to use LITMUS$^{\mathrm{RT}}$ to schedule a real-time workload.

4.1 Supporting Scheduling Quanta

We first discuss our methods for supporting in Linux *aligned* and *staggered* quanta. Aligned quanta provide a consistent view of time that is convenient when all tasks have periods that are some multiple of the quantum size. However, in EDF-scheduling variants, scheduling decisions (and hence quantum allocations) do *not* always occur at timer interrupts, as is the case with PD^2 and S-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.

Before describing how we achieved different quanta alignments, 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 [6], and the architecture of our Intel-based test platform. Kernel-related information should be relevant through Linux version 2.6.20, the version on which LITMUS$^{\mathrm{RT}}$ was developed.

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, resulting in a call to `smp_apic_timer_interrupt()`, the local timer interrupt handler. This handler then calls `smp_local_timer_interrupt()`, which calls `update_process_times()`, which results in a call to `scheduler_tick()`, the function that is responsible for making scheduling decisions during this interrupt. Thus, these timer interrupts represent the quantum boundaries for each processor in our system. 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.

Selecting the quantum size. Given that we require task periods to be multiples of the quantum size, the size of our scheduling quantum should be reasonably small. In our case, we chose the highest natively-supported timer frequency, 1000 Hz, resulting in a quantum size of 1 ms. Experimentation with higher timer frequencies resulted in an unstable system, and require a more in-depth study to determine the feasibility of their use.

Supporting quanta alignments. We supported aligned and staggered quanta by aligning local timer interrupts across processors as follows.

- After initializing local timers normally at system boot, each processor waits for some number of local timer interrupts to be generated before attempting to align quanta. This allows various parts of the system to initialize and stabilize prior to quantum alignment. We conservatively wait for thirty seconds before attempting to align interrupts. We achieved this by adding a check within the Linux local timer interrupt handler, `smp_local_timer_interrupt()`, that calls our function `synchronize_quanta()` when enough interrupts have been generated.

- Within our `synchronize_quanta()` call, the local timer for the calling processor is disabled (by calling the Linux function `disable_APIC_timer()`), and the calling processor waits at a barrier. This barrier is implemented through the use of a variable of type `atomic_t`, so that concurrent reads and increments are performed correctly. Each processor may only pass through the barrier once all processors have reached it. As a result, all processors will pass through the barrier at the same time.

- If *staggered* quanta are desired, then each processor will delay for an additional amount of time immediately after passing through the barrier, so that quantum boundaries on different processors will be evenly distributed over time. This time is equal to the logical CPU identifier of the processor (the result of calling `smp_processor_id()`) multiplied by the quantum size (1 ms) divided by the number of processors. For example, with a 1-ms quantum size and four processors, some processor (ideally) reaches a quantum boundary every 250 μs.

- Finally, each processor restarts its local APIC timer by calling the Linux functions `__setup_APIC_LVTT()` and `enable_APIC_timer()`.

Staggering delays were realized using a non-timer-based kernel delay function called `udelay()`, which is implemented using a software loop with microsecond granularity.

The result of this method is that timer interrupts are either aligned or staggered, as required. A boot option allows us to specify whether aligned or staggered quanta should be provided. Note that we can get aligned quanta even if quanta were substantially misaligned before using this method. Such a statement cannot be made about standard Linux. Also, note that other (non-timer) interrupts cannot interfere significantly with this method of aligning or staggering quanta, since it relies on barriers, and 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 a *global* timer interrupt distributed with interprocessor interrupts (IPIs), and various proposed patches.)

Processor sleep states and interrupts. Some Intel processors may enter sleep states where local timer interrupts are not reliably generated by the local APIC timers, if they are generated at all. In such cases, the method described above would not be a reliable way of providing aligned quanta. To generate local timer interrupts more reliably in this case, Linux disables the local APIC timers for such processors and instead broadcasts a global interrupt (generated by a single external timer source) through the use of IPIs. These interrupts then result in the appropriate calls of the `smp_apic_timer_interrupt()` interrupt handler. While this fix generates interrupts more reliably, it is problematic for our implementation of aligned/staggered quanta, since the time at which IPI signals arrive at each APIC is not much more predictable than the times at which local timer interrupts are invoked at each processor without our method. While the processors that are part of our test platform do not contain the offending sleep states, allowing us to avoid this issue, it highlights an important point. That is, we will need to be able to support aligned quanta in systems where local timers are unreliable or non-existent. Such is likely to be the case for many multicore platforms.

Future directions. In the near future, we plan to begin the next major development cycle of LITMUS$^{\mathrm{RT}}$ during which LITMUS$^{\mathrm{RT}}$ will be ported to two new architectures (Intel and Sun multicore platforms) and the most recent version of the Linux kernel. Starting with kernel version 2.6.22, timer interrupts are no longer generated at a static frequency, and the impact of this change will be explored at this time. Additionally, depending on the timer hardware provided with the Intel- and Sun-based machines that we plan to acquire, certain details could change and require substantial changes to the implementation described here.

4.2 Scheduler Plugin Implementation

We now discuss the details of how we support scheduling algorithms within LITMUS$^{\mathrm{RT}}$ by considering an example, namely our plugin for G-EDF. Most of the scheduling decisions in this plugin are made in the tick handler function `gedf_scheduler_tick()` and the scheduling function `gedf_schedule()`, both of which are considered in detail below. (These two functions implement the `scheduler_tick()` and `schedule()` methods described earlier in Section 3.5 when discussing the plugin interface.) The G-EDF plugin uses the `sched_trace()` framework described earlier to export the execution history to user space (if enabled in the kernel configuration).

State information. The G-EDF scheduler uses the real-time domain abstraction supplied by the LITMUS$^{\mathrm{RT}}$ core infrastructure, parametrized with the EDF order function. Since G-EDF is a global scheduling algorithm, there is exactly one real-time domain for the whole system that manages runnable and to-be-released tasks. Further, G-EDF maintains per-processor state that tracks whether a processor is currently executing a real-time task, and if so, that task's current deadline. In addition, a per-processor `will_schedule` flag is maintained to indicate that a processor is about to reschedule (*i.e.*, it is about to invoke the scheduler). This flag is used to avoid the repeated sending of IPIs to initiate rescheduling across processors in cases where a processor has concurrently determined that rescheduling is necessary.

Preemption check. Each processor's state information is stored in a struct that includes the deadline of its currently-running task (if there is no such task, the deadline is taken to be infinite), and these structs are organized collectively in a linked list that is sorted in increasing deadline order. This list, called the "processor queue," is used to quickly determine whether preemptions are necessary on job releases. The preemption-check function `gedf_check_resched()` (registered as part of the real-time domain initialization), which is called whenever a task has been added to the ready queue, only needs to compare the deadline (which may be infinite) stored for the last processor in the processor queue with that of the highest-priority task in the ready queue (the first task in the queue) to determine whether preemptions are necessary. If preemptions are determined to be necessary, then `gedf_check_resched()` sends IPIs to the appropriate processors to cause rescheduling to occur.

Scheduler tick. When the function `gedf_scheduler_tick()` is invoked on some processor, it updates the budget of the currently-running real-time task (if there is one) on that processor. It also checks the global real-time domain for new job releases.

Since the Linux scheduler tick routine is bypassed for real-time tasks, the management of processor time budgets is delegated to the scheduler plugins. In the G-EDF plugin, the budget of a task is decreased each time it incurs a scheduler tick, which is similar to the stock Linux scheduler (in version 2.6.20).[5] When the budget of the currently-running real-time task is exhausted, a preemption is initiated by returning `FORCE_RESCHED` to the LITMUS^{RT} core tick handler. Further, the processor's `will_schedule` flag is set, as is a flag in the task control block that indicates a job completion.

[5] Linux version 2.6.23 changes this behavior to timestamp-based accounting with nanosecond resolution. We intend to update LITMUS^{RT} to use the same—much more accurate—approach in a future release.

If the system is in real-time mode, then the real-time domain is also checked for new job releases after acquiring the release-queue lock. In the case that a job has to be released, the ready-queue lock is also acquired and the corresponding task is transferred from the release queue to the ready queue. This triggers the preemption-check function as discussed above. If a preemption is required due to the job release, then the lowest-priority processor is forced to reschedule.

Task selection. The function `gedf_schedule()` is invoked whenever a new real-time task needs to be selected for execution. If it is invoked upon a job completion, then some ready task (if one exists) will be selected to execute. However, in other cases, it is possible that a new task is not selected because the currently-running tasks have higher priority than any other ready task. To check for job completions (either determined by the scheduler tick function or signaled by the completing task using one of the job-control system calls described in Section 3.6), the real-time flag field in the task control block is consulted. If it is set to `RT_F_SLEEP`, then the completing task is prepared for its next job release by advancing its deadline and release fields appropriately.

If the system is currently in real-time mode, then the currently-scheduled real-time job (if there is one) on the processor where `gedf_schedule()` is invoked is examined. If it is the highest-priority pending job, then no change is required and `gedf_schedule()` returns the currently-scheduled task. Note that this scenario may occur when a task experiences significant tardiness.

On the other hand, if a preemption is required, then the highest-priority task is dequeued from the ready queue and selected as the next task to be scheduled on the local processor. To actually schedule the new task, it is inserted into the Linux run queue. Correspondingly, the prior task is removed from the Linux run queue. Finally, the position of the processor's entry in the processor queue is updated to reflect the new deadline.

Note that, if a preemption occurs, then the preempted task will not be requeued until the context switch has been completed. The delayed reinsertion into the real-time domain is performed in the function `gedf_finish_switch()` (which is an implementation of the `finish_switch()` method mentioned in Section 3.5).

Other methods. G-EDF provides the `gedf_prepare_task()` function to add newly-created real-time tasks to the current task set. Upon initial arrival, a task's Linux scheduling priority is set to the maximum static priority, its state is set to `TASK_RUNNING` (recall from Section 3.6 that it was created in state `TASK_STOPPED`), and its first job's release time and deadline are initialized. After initialization, it is added to the release queue. G-EDF does not have to handle blocking tasks since all blocking is implemented using native Linux routines, which ensure that any currently-executing real-time task has been removed from the run queue by the time the plugin's method is invoked. Similarly, there is no need to perform per-task tear-down operations since no resources are allocated for individual tasks. The G-EDF mode-change handler reinitializes the per-processor state and prepares new job releases of all real-time tasks to occur synchronously ten milliseconds after the mode change. The functionality needed to provide proper support for the `sleep_next_period()` system call is provided by the LITMUS^{RT} core function `edf_sleep_next_period()` since no scheduler-specific behavior is required.

4.3 User-Space Libraries

As mentioned before, the system call API is not intended to be used directly. Instead, real-time tasks should use the two user-space libraries `liblitmus` (real-time task creation and control) and `libso` (shared objects) that abstract low-level kernel operations.

`liblitmus`, the task creation and control library, provides wrappers around all system calls and also provides convenience functions that perform typical initialization tasks such as locking a newly-created task's virtual memory into the system's memory to avoid page faults and registering the location of the task's non-preemptive section flag. Further, it also provides some utilities intended to be used in shell scripts, most notably `rt_launch`, which can be used to start arbitrary programs as real-time tasks.

The real-time shared object library, `libso`, uses the synchronization services provided by LITMUS^{RT} and the Linux system call `mmap(2)` to provide the abstraction of FMLP-controlled shared objects as well as process naming and in-object memory management. Short resource group locks are implemented in `libso` by using the MCS queue lock algorithm [24] together with the flag-based mechanism described earlier in Section 3.6 to signal non-preemptive sections.

5 Conclusion

In this paper, we have presented an overview of LITMUS^{RT}, an extension to Linux that we have developed to support real-time workloads on multiprocessor systems, and we discussed its current implementation status. At the heart of LITMUS^{RT} is a scheduler plugin interface that allows new scheduling algorithms to be added in a reasonably straightforward manner—additional subsystems of importance include user-space libraries that support the creation and execution of real-time workloads, as well as real-time task synchronization. Given current trends in processor design, and recent interest in providing real-time support within Linux, we believe that this work is timely and could provide a foundation on which other researchers and developers can create and empirically evaluate multiprocessor real-time scheduling and synchronization approaches within Linux.

Future work. A number of directions exist for extending LITMUS^{RT}. First, we want quantum alignments to be re-synchronized periodically, so that alignments are corrected should they become out-of-sync. (On our platform, in our experience, this is not a common occurrence, but it could be on other platforms.) Second, we want to provide support for finer-grained locking in global schedulers. Third, we want to reduce scheduling overheads in global algorithms so that they are logarithmic or constant in the number of processors or logical CPUs, instead of linear, as they are now. This will become especially important as the number of logical CPUs (cores or hardware threads) on a platform increases. Fourth, there are several parts of the implementation that need to be debugged or made more robust to erroneous use. Fifth, we wish to use Feather-Trace as a mechanism to improve the efficiency of the `sched_trace()` function that we provide to facilitate scheduler debugging and evaluation. Finally, we are planning a large development cycle in which we will port LITMUS^{RT} to both the most recent kernel version, and two multicore architectures: a quad-processor Intel machine consisting of quad-core chips (16 total cores), and the Sun UltraSPARC T2 (Niagara 2), which consists of eight cores with eight hardware threads per core (eight total cores, 64 total hardware threads). Our goal is to evaluate both LITMUS^{RT} and various scheduling approaches in environments with shared caches and higher core counts. This development cycle will also consider issues related to significant changes to the Linux scheduler, and achieving the quantum alignments desired in a kernel with a substantially different timer interrupt infrastructure, on different timer hardware.

References

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

[2] T. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, 1991.

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

[4] A. Block, B. Brandenburg, J. Anderson, and S. Quint. Feedback-controlled adaptive multiprocessor real-time systems. In submission, 2007.

[5] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 71–80, 2007.

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

[7] B. Brandenburg and J. Anderson. Feather-trace: A lightweight event tracing toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'07)*, pp. 19–28, 2007.

[8] B. Brandenburg and J. Anderson. Integrating hard/soft real-time tasks and best-effort jobs on multiprocessors. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pp. 61–70, 2007.

[9] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In submission, 2007.

[10] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 101–110, 2007.

[11] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pp. 111–123, 2006.

[12] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pp. 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[13] C. Chen and S. Tripathi. Multiprocessor priority ceiling based protocols. Technical Report CS-TR-3252, Univ. of Maryland, 1994.

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

[15] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. *Real-Time Systems*, to appear.

[16] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pp. 75–84, 2006.

[17] P. Gai, M. di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time And Embedded Technology Application Symposium*, pp. 189–198, 2003.

[18] H. Hartig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*, 1998.

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

[20] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, 1993.

[21] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[22] P. McKenney. Attempted summary of "RT patch acceptance" thread. http://lkml.org/lkml/2005/6/7/256, 2005.

[23] P. McKenney. Shrinking slices: Looking at real time for Linux, PowerPC, and Cell. http://www-128.ibm.com/developerworks/power/library/pa-nl14-directions.html, 2005.

[24] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[25] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, 1991.

[26] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

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

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

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

[30] V. Yodaiken and M. Barabanov. A real-time Linux. *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, 1997.