Joint Opportunities for Real-Time Linux and Real-Time Systems Research

Björn B. Brandenburg
Department of Computer Science
The University of North Carolina at Chapel Hill
Campus Box #3175, Sitterson Hall, Chapel Hill, NC 27599-3175
bbb@cs.unc.edu

James H. Anderson
Department of Computer Science
The University of North Carolina at Chapel Hill
Campus Box #3175, Sitterson Hall, Chapel Hill, NC 27599-3175
anderson@cs.unc.edu

Abstract

Based on lessons learned in developing and maintaining LITMUS$^{\mathrm{RT}}$, arguments in favor of an intensified collaboration between the academic and the open-source real-time communities are presented, and several ways in which ongoing efforts in these two communities may benefit each other are outlined. Some (unfortunately) commonly-encountered sources of friction and mutual misconceptions, which result from differing backgrounds and objectives in these two communities, are exposed, with the goal of finding common ground. Further, a "wish list" is presented of possible additions and changes to PREEMPT-RT that would enhance Linux's viability as the "platform of choice" for real-time-systems research. These improvements are substantiated by examining EDF-HSB, a candidate algorithm for earliest-deadline-first (EDF) scheduling support in Linux that integrates hard and soft real-time guarantees as well as best-effort scheduling with bandwidth reservations.

1 Introduction

Due to its openness and its applicability across a wide range of hardware platforms, Linux has been popular with researchers for many years [47]. In research on real-time systems in particular, Linux has served as a starting point for numerous projects [19, 21, 24, 33, 37, 44, 46, 48]. Given these strong ties to real-time-systems research, one could reasonably expect Linux to provide many real-time-related features that are based on scheduling and synchronization algorithms that are provably among the best approaches (from an analytical standpoint). Sadly, this is not the case.

Linux's central real-time features focus on low interrupt latency, static priority scheduling, and basic priority inheritance [30, 41]; while undoubtedly important, these features reflect the state of the art of over 20 years ago. Moreover, recently-added, more advanced features, such as hierarchical scheduling and bandwidth limiting, are *not* rooted in analytically-sound foundations [6]. In essence, there is a severe disconnect between academia and Linux technologists—few (if any) of the advances made in projects based on Linux have found their way back to Linux itself.

This lack of communication and exchange of ideas is very regrettable when in fact academics and Linux developers increasingly need each other. With ever-increasing hardware complexity, academic researchers critically depend on a strong foundation—building a fully-featured OS from scratch is prohibitively expensive. Given Linux's extensive multiprocessor support (since version 2.6) and the widely-acknowledged, yet still growing importance of multicore architectures, Linux will likely remain the primary platform for applied systems research in the foreseeable future. At the same time, multicore architectures pose complex multiprocessor resource-allocation and optimization challenges for which "trial-and-error" design yields only diminishing returns. While slow-moving at times, academia can offer insights and advice on fundamental algorithmic tradeoffs that are only rarely exposed on mailing lists.

Hence, we strongly believe that improved collaboration between the Linux and academic real-time communities would be mutually beneficial. As a first step towards that goal, this paper comments on some of the common barriers and misconceptions that have hindered cooperation in the past, and examines real-time concepts and interfaces that would benefit both researchers and practitioners if implemented in Linux.

Defining "Real-Time Linux." Many designs—often quite different from one another—have been labeled "real-time Linux" in the past.

Real-time Linux variants can generally be categorized into two groups. In a *native* design, the Linux kernel is the only kernel present and responsible for meeting real-time requirements, and real-time tasks are regular Linux processes. In contrast, in a *para-virtualized* design, a specialized (hard) real-time-capable microkernel (or hypervisor) is conceptually inserted between Linux and the actual hardware. Such implementations follow a classical microkernel design [28] in which Linux takes over the role of an OS server and is scheduled as a background, non-real-time thread by the microkernel (e.g., [24, 48]). Real-time tasks are specialized threads (i.e., not Linux processes) that are directly dispatched by the microkernel.

Because of Linux's beginnings as a traditional monolithic kernel with (in the context of real-time-systems design) excessively-long non-preemptive sections, early Linux-based real-time systems were commonly based on para-virtualization. There are two key advantages to such a design. First, low interrupt latencies can be guaranteed to real-time tasks regardless of any deficiencies in the Linux kernel. Second, only (relatively) small changes to the Linux kernel are required, which means that integrating improvements made in newer Linux versions is (relatively) easy. A good example for these benefits is the L4Linux/Fiasco system [24], in which Linux is para-virtualized on top of Fiasco, TU Dresden's L4-based microkernel: initially released in 1996 for Linux 1.3.94 [25], L4Linux is still reliably tracking the latest Linux kernel versions in 2009.

Unfortunately, para-virtualized real-time Linux variants suffer from two significant drawbacks: **(i)** as mentioned above, real-time time tasks are not Linux processes and **(ii)** there is only little benefit to mainline Linux. Regarding (i), to ensure predictably low interrupt latencies, real-time tasks usually execute directly on top of the microkernel and cannot make use of Linux services (such as device drivers, POSIX IPC, synchronization primitives, filesystems, etc.). This limitation is fundamental since para-virtualization does not improve Linux's real-time capabilities; rather, it enables real-time tasks to safely coexist with the Linux kernel. (One common way to enable communication between real-time threads and Linux processes in such systems is the use of non-blocking queues and buffers [3, 4].) Compared to ordinary Linux-based development, para-virtualized designs pose a greater engineering challenge due to the unusual and more complex system architecture and the restricted runtime environment for real-time tasks.

Regarding (ii), para-virtualization fundamentally takes the existing Linux kernel as a given; it is focused on engineering a solution that circumvents Linux's deficiencies rather than solving them. As such, the capabilities of the Linux kernel are not improved and only few (if any) patches are conveyed back to mainline Linux. As an example, consider RTLinux [48]: despite both its use in industrial embedded systems and having been supported commercially for more than ten years, it has had very little influence on mainline Linux. In some sense, para-virtualization generates a "two-class society:" first-class applications, i.e., those that are important enough to warrant the effort of developing a custom para-virtualized system, can rely on reliable real-time properties; however, second-class applications (everything else) cannot. Note that the latter includes desktop applications such as media playback.

To summarize, while para-virtualization may be the only feasible (Linux-based) design for applications with very stringent latency requirements (e.g., engine control software), a native design is generally preferable for the vast majority of applications *if timing constraints can be met*: from the point of view of commercial users of real-time Linux, such a design can result in significant time-to-market and cost savings (due to a familiar development environment, greater talent pool, code re-use, etc.), and from the point of view of the Linux community, there is an increased chance that improvements are contributed back and that they can be integrated into mainline Linux.[1]

Hence, we believe that the goal of Linux-based real-time research should be to improve native real-time Linux so that it becomes a viable solution for the widest range of applications possible. Consequently, we do not consider para-virtualized designs in the rest of this paper, and our proposals concerning "real-time Linux" should mostly be understood in the context of the PREEMPT-RT patch set [39].

[1]We acknowledge that microkernel designs can also be attractive for non-technical reasons, e.g., they allow circumvention of Linux's licensing requirements by placing proprietary code in a separate address space. Such concerns are beyond the scope of this paper.

2 Sources of Friction

With their foundations firmly grounded in openness and the free flow of ideas, the open source community and academia have many more commonalities than differences. Nonetheless, certain issues have repeatedly caused some degree of confusion and mutual misunderstanding. The following list of topics is neither exhaustive nor can it be "representative" of the points of view encountered in such large and diverse communities. However, we hope that future cooperation can be improved by pointing out how differences in background and objectives have led to misconceptions in previous interactions between these two communities.[2]

Low latency vs. predictability. A recurring debate is the topic of "real-time vs. real-fast" [34]: in practice, real-time constraints are often phrased in terms of latency requirements, whereas the academic definition of a real-time system does not even mention latency.

Formally, real-time systems are distinguished from other computing systems by having a *dual* notion of correctness: proper behavior depends not only on *logical* correctness ("it does the right thing"), but also on *temporal* correctness ("at the right time"). Note that, as with all formal properties, both correctness criteria require rigorous proof. Hence, real-time research is the study of systems with *provable* temporal properties: systems for which we can predict runtime behavior with mathematical certainty (if all assumptions are met).

In practice, it seems that "real-time" is often equated with an OS's ability to transfer control to real-time tasks as quickly as possible, maybe together with a bound on worst-case latency. This is an over-simplification of the notion of "real-time computing." Obviously, latency crucially impacts what kind of guarantees can be made, but even a system with negligible observed latency is not truly a real-time system if it is constructed from algorithms that do not lend themselves to formal analysis; such a system is merely "fast."

Another frequently-encountered difference is that, in practice, system design is often focused on supporting *one* dedicated real-time task, whereas a proper real-time system should support any number of such tasks (as long as meeting their collective timing constraints is feasible[3]).

If operating system design is predominantly focused on providing low latency to *the* highest-priority task (as opposed to meeting a set of specified timing constraints and enforcing budgets—see Sec. 3), then composing a real-time system from multiple tasks becomes unnecessarily complicated, if not impossible (e.g., given possible starvation, should a real-time web service's priority be higher or lower than the network stack that it depends on?). As a result, design patterns such as "select/poll-based master loops" and "one real-time task per core" are often favored even though designs that employ multiple cooperating real-time tasks would be more desirable in terms of reliability and achievable resource utilization (i.e., more could be done with fewer processors).

Of course, we do not argue that real-time Linux should not provide low worst-case latencies. However, we do argue in favor of phrasing the discussion in terms of deadline and jitter requirements of supported workloads instead of raw (and often, it seems, somewhat arbitrary) latency numbers. Further, we believe that the kernel should support a more expressive task model and make scheduling decisions at runtime based on task constraints (as opposed to relying on a single static priority value—see Sec. 3).

Proofs and verification. As mentioned above, claims of real-time properties require an accompanying proof of correctness. Of course, in practice, even proving logical correctness is done rarely at best. Ignoring the question of whether it is beneficial to verify every program, it is probably safe to assume that many embedded development efforts are under sufficiently-pressing time-to-market and cost constraints that formal verification would likely be skipped even if tools for computer-aided correctness proofs were readily available. Worse, in reality, we are, in all likelihood, decades away from being able to formally verify software systems of Linux's complexity.

Does this make real-time research "impractical"? Does it mean that real-time-system design practices in general, and Linux in particular, have little to gain from current research? Certainly not!

Even though Linux itself is not formally verified,[4] it is built around concepts and algorithms that have been thoroughly explored in research papers. Examples are numerous: from data structures such as red-black trees and heaps to TCP's congestion-avoidance protocol, from pattern-matching algorithms to concepts such as fairness, from encryption to mandatory access control, Linux's internals rely on building blocks for which desirable properties have been proven in the research literature. Why should Linux's real-time features be any different?

We argue not that every system has to be proven to be correct (which, though desirable, is clearly impractical at this point), but that every system should be *composed*

[2]The authors acknowledge that their point of view is naturally biased towards academia.

[3]Establishing the feasibility of timing constraints lies at the core of real-time analysis. A thorough discussion of feasibility is beyond the scope of this paper; however, the interested reader is strongly encouraged to explore this important topic [31].

[4]Interestingly, growing interest in tools such as the *Clang Static Analyzer* [32], the *Coverity Integrity Checker* [20], and *Sparse* [42] seems to indicate that there is significant demand for static program analysis techniques.

from parts that *can be* and *have been* proven to be correct. The implemented algorithms should never be the weakest link.

Overheads. There has also been some controversy regarding the treatment of overheads in the academic literature. Specifically, concerns have been voiced that published scheduling analysis is not applicable to real systems because overheads are routinely assumed to be negligible.

Assuming overheads to be negligible is indeed standard practice in real-time analysis and may appear to be an oversimplification on first sight. However, this is not the case. In work on real-time resource allocation, overheads are often ignored in the primary analysis, as this reduces notational clutter that can obscure the results. Once such analysis has been obtained, overheads can be accounted for by inflating task execution costs. Thus, what we see here is actually a classic separation of concerns: establish an algorithm's essential properties first, and then factor in overheads later.

Mode of operation. Last, but not least, there is a fundamental difference in how academia and the Linux community function.

The modus operandi of the Linux community is captured well by the widely-known mantra "release early, release often." Developers are encouraged to provide early versions of their work to the *Linux kernel mailing list* (LKML) to solicit feedback and advice on what is often still a rough and unfinished prototype, and participants may engage in lively discussions before much developer effort has been wasted on potentially futile approaches—but also, sometimes, before the problem and possible solutions are sufficiently well understood. Many projects go through frequent and fast iterations, and the resulting traffic on LKML can easily become overwhelming to observers.

In contrast, the academic culture is mostly focused on work that is, to some degree, finished and polished. The bar to publication can be high and the academic peer reviewing process is an anonymous, mostly one-way conversation, and its progress is usually measured in weeks and months, not hours or days. Traditionally, conferences provide the main venue for discussion, and as such, convergence on key research directions tends to happen through in-person exchanges. In discussions, academics tend to shy away from quick and categorical answers.

With such diverging cultural norms, mutual misunderstanding is all but guaranteed to occur. The intensity of the Linux community's interactions can appear daunting and the brief messages exchanged during quick back-and-forth discussions may come across as being ill-prepared. Conversely, academics can appear to be slow moving, doubtful, and hesitant, characteristics that, when combined, are easily mistaken as disinterest. In the interest of improved collaboration between these communities, it is best to be aware of these differences.

3 Extending Real-Time Linux

Having touched on some of the issues that can prevent effective collaboration in the previous section, we next discuss some design choices, algorithmic concepts, and research directions that we believe could serve as starting point for enhancing real-time Linux.

3.1 Design Space

First, we discuss how real-time Linux could be structurally improved to both better support applied research and adapt to specific use cases.

Support scheduler plugins. The topic of scheduler plugins is a contentious issue and has provoked intense discussions in the Linux community in the past. While preferred by most scheduler developers (perhaps unsurprisingly), the concept has been rejected by other influential members of the Linux community. One frequently-voiced criticism of scheduler plugins is that they would likely lead to a proliferation of special-purpose plugins, and that Linux as a whole would be better served if all interested parties came together to build a "universal scheduler" that meets all requirements [35].

Unfortunately, such a scheduler is unlikely to exist. Evidence for this viewpoint can be found in recent implementation studies [17, 19] involving several widely-studied multiprocessor real-time scheduling algorithms. In these studies, none of the tested algorithms proved to be the "best;" rather, each algorithm performed well for some of the considered workloads, but not so well for others.

As an analogy, consider filesystems: there exist filesystems specialized for high-performance computing (HPC) and filesystems for embedded flash storage. It is widely acknowledged that there exist good reasons for the development of specialized filesystem implementations, and it is considered acceptable both to divide developer effort among multiple filesystems and for administrators to choose an appropriate filesystem for their workload. In light of this, why should one expect the same scheduler to be appropriate for both HPC and embedded real-time systems?

Hence, we argue strongly in favor of introducing a scheduler plugin interface in Linux. Not only would this greatly simplify the prototyping and experimental evaluation of new scheduling algorithms, it would also allow embedded Linux variants to be tailored to their workloads.

Note that, ideally, the default CFS/static-priority scheduler should also be a plugin (so that it can be replaced), and that appropriate run queues should be provided and *encapsulated* by each plugin (as opposed to passing pointers to a statically-allocated run queue to plugin-provided callbacks). The rationale for encapsulating the implementation of run queues is that there are many different priority-queue implementations that are worth exploring [15], and the current run-queue implementation is inherently limiting, as is discussed next.

Enable global scheduling. Research has shown that there are fundamental differences between partitioned (per-processor run queues) and global (shared run queue) scheduling. In particular, *global earliest-deadline first* (G-EDF) has been shown to have properties that are desirable for certain classes of soft real-time systems [22]. Unfortunately, implementing global schedulers in Linux, while feasible (see for example UNC's LITMUS$^{\text{RT}}$ [45]), is unnecessarily complicated because Linux's run queues are inherently partitioned. Since the run-queue locks also serve to serialize updates to process state, the standard run queues cannot simply be circumvented. As mentioned above, the run-queue implementation should be considered part of a scheduling plugin's internals and not statically mandated for all schedulers.

Regarding the frequently voiced objections to G-EDF's viability in a "real" system, it should be noted that *xnu*, the kernel underlying Apple's multimedia-friendly OS X, has been relying on G-EDF to support real-time applications on multiprocessors for several years [5].

Support dynamic-priority scheduling. In recent years, interesting variations of G-EDF that reduce average deadline *tardiness*, i.e, the amount of time by which a deadline is missed, have been proposed [7, 26]. These variations fall into the class of *dynamic-priority* algorithms because a job's (see Sec. 3.2 below) priority may change at arbitrary times. Similarly, the only optimal[5] multiprocessor scheduling algorithm for which (published) implementations exist [17, 19, 43] also belongs to the class of dynamic-priority algorithms (as all optimal multiprocessor scheduling algorithms must).

Yet another class of algorithms of considerable recent interest, so-called *semi-partitioned* schedulers [2], includes several dynamic-priority variants. Under semi-partitioned scheduling, each task is categorized as either fixed (executing on one processor only) or migrating (executing on several processors). Such algorithms allow for better system utilization than pure partitioning approaches, without the runtime overhead of allowing all tasks to migrate.

So that the desirable properties of these algorithms can be exploited, Linux's future scheduler plugin interface should be sufficiently flexible to support the implementation of algorithms in which processes may change priority and migrate frequently (and at arbitrary points in time).

Throw out POSIX. The real-time POSIX standard is mostly outdated and does not apply particularly well to multicore systems. Research plugins and scheduler plugins targeting embedded systems with pre-determined workloads should not be constrained by such legacy requirements.

For example, consider one of the most-widely used soft real-time applications: media playback and interactive graphical user interfaces (such as in games). Currently, real-time priorities require superuser privileges, hence it is not possible (nor, given static-priority scheduling without budget enforcement, particularly advisable) for non-privileged users to make such applications proper real-time processes. Ideally, future desktop and windowing systems should be able to provide real-time capabilities in a controlled fashion—this requires interfaces and abstractions that were not envisioned by the POSIX standard; hence, scheduler plugins should not be artificially restricted in such ways.

3.2 Task Model and Predictability

Instead of solely relying on static priorities, real-time Linux should offer a more expressive API that explicitly allows the specification of timing constraints. While there are many ways of specifying timing constraints, we advocate that initial support should focus on the sporadic task model since it is the most widely-studied such model (in work on multiprocessor real-time scheduling).

In the sporadic task model, a task $T_i = (p_i, e_i, d_i)$ is characterized by three parameters: its *minimum inter-arrival* time p_i, also known as its *period*, its *worst-case execution cost* e_i, and its *relative deadline* d_i. The recurring activation of T_i is modeled as a sequence of job releases $T_i^1, T_i^2, \ldots$, subject to the following constraints: job releases are separated by at least p_i time units, each job requires at most e_i time units to complete execution, and each job should complete within d_i time units after its release. Note that a sporadic task is just a logical entity that is not scheduled; instead, jobs are scheduled by the OS.

It is not immediately clear how to best apply the sporadic task model in the context of a UNIX-like process model.[6] If jobs, and not sporadic tasks, are scheduled, then how should sporadic tasks be mapped onto Linux

[5]See [22, 31] for a proper definition of "optimal."

[6]To avoid confusion, in the subsequent paragraphs, we use "task" to denote a sporadic task, and "process" to denote Linux's notion of an executing program instance.

processes? Further, discovering the *actual* worst-case execution time of non-trivial code paths in Linux is effectively impossible on contemporary processors given current worst-case execution time analysis techniques. Is the sporadic task model therefore "impractical"?

The sporadic task model should be seen as reservation mechanism in the context of a UNIX-like OS such as Linux.[7] In this interpretation, a sporadic task (associated with a process) is simply an execution-time budget with a minimum separation between replenishments and a guarantee that the budget can be consumed within a specified time window (as defined by the task's relative deadline) after each replenishment. The real-time scheduler should hence ensure that all budgets can be consumed before their corresponding deadline, and that individual processes do not consume more time than their specified budget. The real-time guarantee provided by such a system is that, if a real-time application stays within its allocated budget, then it will meet its application deadlines. Because budgets are enforced, this is even true if some applications misbehave (e.g., enter an infinite loop), a concept known as *temporal isolation*.

In the real-time literature, the mechanism described above is sometimes referred to as a *polling server*, and the concept of controlling the execution of one or more processes by means of a budget accounting policy is generally called a *server abstraction* (this corresponds to Linux's "cgroup" abstraction).

Server-based approaches hold the promise of controlling interference caused by (threaded) interrupt handlers [8, 27]. Similarly, several server schemes have been proposed in the literature to reserve bandwidth for non-real-time processes (e.g., see [1, 9, 12, 37]). Since scheduling analysis exists for most server schemes, server-based interference control offers the ability to compute budgets such that interrupts or background workloads are processed quickly, but without unduly delaying real-time applications.

To summarize, server schemes offer analytically sound ways of compartmentalizing a system at runtime; we advocate that Linux's cgroup support should be extended to enable the implementation of various server schemes in scheduler plugins.

When realizing the sporadic task model as outlined above (and some of the server schemes), it must be ensured that the implemented budget accounting meets the assumptions of published scheduling analysis. This is especially true when dealing with suspensions, e.g., when processes block on I/O operations or due to semaphore contention. Accounting for suspension times accurately is a notoriously hard and largely unsolved problem (see [40] for details). Hence, most analysis currently requires job suspension times to be included in the worst-case execution time (static-priority scheduling is a notable exception to this). Consequently, real-time processes should consume budget even while they are suspended, otherwise suspensions can cause scheduling anomalies.

A more expressive real-time API should also allow policies to be specified for handling budget overruns and underruns. For example, LinuxRK [37] supports a task model in which real-time processes that overrun their budget can compete for execution at non-real-time priorities; the real-time budget is interpreted as a minimum supply in this case. Similarly, the BACKSLASH algorithm [29] allows tasks to "borrow" against future allocations. Due to the pessimism typically involved in deriving budget requirements for real-time tasks, most real-time applications, most of the time, will only partially use their allocated budgets. Such budget underruns create *dynamic slack*, which itself can be scheduled and used to reduce the impact of overruns in other applications or to improve the response time of non-real-time workloads (for examples, see M-CASH [36] and Sec. 4 below).

3.3 A Prototype

To facilitate applied real-time scheduling research and to evaluate the aforementioned concepts (and others) on real hardware platforms, our group has developed LITMUS$^{\mathrm{RT}}$, the **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems [16, 17, 19, 45]. In this section, we briefly summarize some of the lessons learned while building and maintaining LITMUS$^{\mathrm{RT}}$.

LITMUS$^{\mathrm{RT}}$ predates CFS's scheduler-class hierarchy and thus was originally designed to hook manually into the Linux scheduler. We were delighted when the CFS patches introduced the concept of proper scheduling classes, as this promised to provide a much cleaner method for LITMUS$^{\mathrm{RT}}$ to interface with the core scheduler. However, the scheduler-class interface turned out to be too restrictive to serve as the sole LITMUS$^{\mathrm{RT}}$ interface (in no small part due to Linux's reliance on partitioned run queues). Hence, LITMUS$^{\mathrm{RT}}$ still hooks into several scheduler routines (but to a lesser extent than before).

Migrations. To avoid races with code paths trying to resume processes from a suspension (`try_to_wake_up()`), every runnable process in Linux needs to be on one of the per-processor run queues at all times. Since one of the primary features of LITMUS$^{\mathrm{RT}}$ is the support of global scheduling algorithms, the scheduling path is complicated by the need to migrate processes in `schedule()`. To ensure consistency, process migrations require a processor to acquire both the local and the remote run-queue locks. Because `schedule()` already holds the local run-queue lock, and since run-

[7] Resource reservations have been studied extensively in the context of LinuxRK [37].

queue locks must be acquired in order of increasing lock address (to avoid deadlock), such migrations may require first dropping and then re-acquiring the local run-queue lock. Since process state may change in the brief window in which the local run queue is unlocked, care must be taken to check for conflicting updates after the migration completes. Migrations could be greatly simplified if separate locks were used to protect process state and run-queue integrity.

Task rejection. LITMUS$^{\mathrm{RT}}$ allows scheduler plugins to reject processes from becoming real-time tasks (e.g, if the resulting task system would be infeasible), which requires hooks in `sched_setscheduler()`. Once Linux has a proper scheduler plugin API, changing scheduling classes/plugins should also be modularized, and the acceptance logic should be delegated to scheduler plugins.

Polling servers. Budget enforcement requires the scheduler to split the run queue in two. The *ready queue* contains processes that are runnable and eligible to execute (i.e., the corresponding job is released and has a non-zero budget), and the *release queue* contains processes that are runnable but are awaiting the replenishment of their budgets. In LITMUS$^{\mathrm{RT}}$, there are two implementations of this. In event-driven schedulers (such as G-EDF), job releases are triggered with high-resolution timers, hence the ready queue only conceptually contains tasks. In quantum-driven schedulers (such as PD^2 [43]), the release queue is realized as a circular array of binomial heaps, so that released processes can be efficiently merged into the ready queue at each quantum boundary.

Global scheduler state. LITMUS$^{\mathrm{RT}}$'s G-EDF implementation is based on the concept of *task linking*,[8] under which the assignment of *tasks* that should be scheduled (the "links") is maintained separately from the set of actually-scheduled, backing *processes*. Note that the set of scheduled processes can trail the links.

This has several advantages. First, the implementation of the scheduling policy becomes independent from hardware constraints: while a processor can only physically reschedule its local process (i.e., perform the actual context switch), any processor can re-link any task to any processor. In our experience, this avoids several potential races in the implementation of global policies such as G-EDF. Second, the actual preemption/context-switching logic simply becomes a matter of tracking what process *should* be executing—this code can be policy-independent and should be re-used between plugins. Third, by investigating the links of other processors, it is relatively easy to avoid superfluous migrations and context switches. Fourth, link-based scheduling lends itself to real-time analysis—the system can be understood as consisting of a scheduler that can reschedule any processor at any time, and a scheduling latency that accounts for any delay in actually enacting task-link updates [10]. This nicely abstracts interprocessor interrupt latency, non-preemptive sections, and all other OS delays into a single latency value, which can be accounted for by inflating execution costs (as discussed in Sec. 2).

Synchronization. Based on task-linking, LITMUS$^{\mathrm{RT}}$ includes support for userspace non-preemptable sections, in which interrupts are enabled but task-link changes are not enacted until the currently-scheduled process signals that it has left its critical section. This mechanism can be protected against abuse with a simple timeout mechanism. LITMUS$^{\mathrm{RT}}$ also supports FMLP-based real-time synchronization [10, 13, 14, 18].[9]

4 Case Study: EDF-HSB

In this section, we summarize EDF-HSB [12] as a case study to highlight what we believe to be desirable properties for an integrated scheduling approach for mainline Linux, and how those properties can be achieved.

Design goals. EDF-HSB was designed to support a wide range of workloads. Real systems are likely to consist of mixes of applications with differing timing constraints, e.g., a few applications that are highly sensitive to jitter, several applications with moderate real-time requirements, and some background processes (such as logging, system maintenance, etc.).

Hence, EDF-HSB explicitly distinguishes between three task categories: *hard real-time* (HRT), *soft real-time* (SRT),[10] and *best-effort* (BE) tasks. Note that this task type information is used at runtime, and that one or more of these categories may be absent in any given system.

The design of EDF-HSB hinges on the assumption that the HRT component requires only a small fraction of the total system capacity, an assumption that we believe to be true for the vast majority of real systems (and especially those that are being deployed using Linux).[11]

Conversely, the SRT component is presumed to (potentially) require a large fraction of the system's capacity. Hence, EDF-HSB is carefully designed to avoid both unnecessarily wasting utilization to accommodate HRT tasks and placing a limitation on the maximum total uti-

[8]First described in the context of synchronization, see [10].

[9]A description of the synchronization support is beyond the scope of this paper—please refer to [14] for a discussion of the implementation.

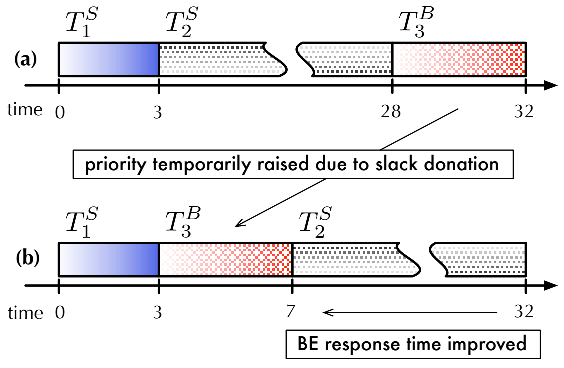
[10]Using the bounded-tardiness definition of "soft real-time," e.g., see [22].

[11]Our assumption has been confirmed repeatedly in discussions with industry colleagues.

lization that can be reserved for SRT tasks (within hardware limits). Note that this cannot be achieved in general with any pure partitioning approach (such as Linux's current scheduler).

Finally, since the HRT and SRT reservations are likely pessimistic due to "engineering margins" and variations in execution time requirements, EDF-HSB has been designed to reclaim unused capacity and redistribute it to "needy" tasks at runtime, i.e., EDF-HSB incorporates dynamic slack scheduling.

Slack scheduling is best explained with an example. Suppose there are two SRT tasks, $T_1^S = (30, 7, 30)$, $T_2^S = (50, 25, 50)$, and one BE task, T_3^B, in the ready queue. By EDF, T_1^S is scheduled first. Now suppose that T_1^S completes its execution after only three time units, i.e., it leaves four time units of its per-job reserve unused. Because real-time analysis of the system showed that all timing constraints are met if T_1^S uses its full allocation, the system can schedule *any* lower-priority work using the unused part of T_1^S's allocation prior to T_1^S's deadline *without the risk of violating temporal correctness.* In this case, if T_3^B requires no more than four time units to complete, then its response time has been lowered by 25 time units, as is shown in Fig. 1.



T_1^S
T_2^S
T_3^B
(a)
time
0
3
28
32
priority temporarily raised due to slack donation
T_1^S
T_3^B
T_2^S
(b)
time
0
3
7
32
BE response time improved

FIGURE 1: *Benefits of slack scheduling.* **(a)** *Without slack scheduling, T_3^B is delayed until T_2^S completes.* **(b)** *With slack scheduling, T_3^B can execute for up to four time units at T_1^S's priority and hence complete before T_2^S is scheduled. This reduces T_3^B's response time by 25 time units.*

In general, slack scheduling is a powerful technique that can significantly reduce the impact of budget overruns and drastically lower response times of background processes.

Algorithm structure. The purpose of this description is to serve as an overview; please see [12] for a full definition of the algorithm. The following paragraphs correspond to Fig. 2, which illustrates EDF-HSB's structure.

Under EDF-HSB, similar to Linux's hierarchy of scheduling classes, the components are statically prioritized: eligible HRT tasks have higher priority than SRT tasks, and SRT tasks have higher priority than BE tasks. However, unlike Linux, special HRT servers are employed to rate-limit the HRT component, polling servers are used to isolate both HRT and SRT tasks, and BE servers are used to reserve bandwidth for BE work at SRT priority.

The HRT component is partitioned across processors to enable the re-use of uniprocessor HRT schedulability analysis. The SRT component is scheduled globally to ensure that the system can be fully utilized. As presented in [12], EDF-HSB also globally schedules the BE component; however, this could be easily changed if so desired.

EDF-HSB uses partitioned EDF together with server-based rate-limiting rules (see [12] for details) to schedule the HRT component. The SRT component is scheduled using G-EDF, and the BE component is scheduled using some non-real-time scheduler (global FIFO in [12], but integration with Linux's CFS is also possible).

Slack scheduling is carried out by collecting records of unused time in a global capacity queue, which is sorted by the donating jobs' deadlines. Capacities are scheduled with SRT priority. When a capacity is selected to "run," it is consumed by a SRT or BE job, which is selected by a heuristic and then scheduled. Such a job continues to consume the capacity until either the capacity is exhausted or higher-priority jobs arrive. Note that unused capacities must be properly "expired," as explained in [12].

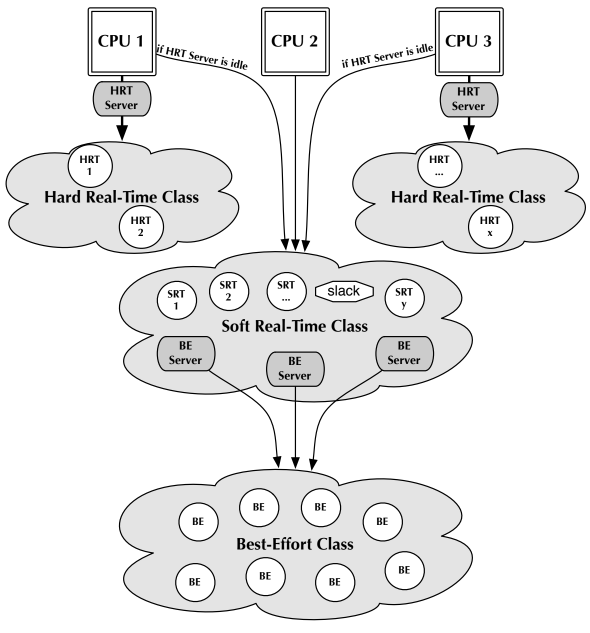


FIGURE 2: *Illustration of EDF-HSB.*

Properties. EDF-HSB has a number of desirable properties. The frequency of HRT interference can be freely adjusted (the period of the rate-limiting HRT server can be chosen independently of HRT deadlines; this allows “performance tuning”), and no over-provisioning of the HRT component is required (e.g., if the HRT tasks on one processor cumulatively require 20% of the processor’s capacity, then the HRT server has a reserved utilization of exactly 20%, and no more).

EDF-HSB also ensures that BE processes are not starved because the SRT-level capacity reserves for BE workloads (if non-zero) ensure progress even if all SRT tasks are backlogged.

In [12], experiments are presented that demonstrate the effectiveness of slack scheduling in improving system responsiveness. In these experiments, slack scheduling resulted in a reduction of measured worst-case response times of BE tasks from over 300ms under G-EDF to less than 75ms under EDF-HSB [12]. To put these numbers into context, typing shell commands in a G-EDF-scheduled system (under a test load) felt notably sluggish, whereas (under the same load) the EDF-HSB-scheduled system exhibited average response times that, for humans, are indistinguishable from an idle system [12]. This can make a major difference in real-world scenarios. Consider, for example, an administrator who wants to re-configure a host that is suffering from overload of its real-time web services: with slack scheduling, the system remains responsive to background commands as long as at least one HRT or SRT task generates slack.

The design of EDF-HSB makes it a viable choice for mainline Linux even if scheduler plugins remain excluded because it is inherently able to support a wide range of real-world workloads.

Open questions. Some extensions of EDF-HSB are still the subject of ongoing work. For one, lock-based synchronization across component boundaries remains a largely unsolved problem. Further, the dynamic nature of Linux workloads require server parameters to change at runtime, a process known as *re-weighting*. A relatively simple solution is to postpone parameter changes (and, consequently, enacting task arrivals and departures) until the beginning of the next period, but better results are attainable [11]. In the context of EDF-HSB, re-weighting schemes suitable for its hierarchical, server-based structure have yet to be developed. Note that task suspensions do not necessarily trigger re-weighting; re-weighting is only necessary on HRT/SRT task creation/destruction.

5 Conclusion

In this paper, we presented arguments in favor of greater cooperation between the real-time Linux community and academia, supported by concrete suggestions for improved real-time support in Linux. Further, we pointed out some common misconceptions that have lead to misunderstandings in the past.

Improved cooperation hinges on the willingness of both communities to approach each other. It is surely unreasonable for the Linux community to expect large, production-quality code contributions from academics. Likewise, it is unreasonable for academics to expect the Linux community to discover all relevant concepts and papers by itself. Instead, as an achievable short-term goal, participants should aim for more discussions and a lively flow of problem descriptions and possible solutions back and forth.

To that end, we suggest that (more) academics should consider attending next year’s Real-Time Linux Workshop (RTLWS) [38], and (more) Linux developers should consider attending next year’s International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT) (to be held July 2010 in Brussels, in conjunction with the EuroMicro Conference on Real-Time Systems [23]).

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 3–13. IEEE, December 1998.

[2] J. Anderson, V. Bud, and U. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 199–208. IEEE, July 2005.

[3] J. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. IEEE, December 1996.

[4] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 28–37. IEEE, December 1995.

[5] Apple Inc. xnu source code. http://opensource.apple.com/source/xnu/.

[6] T. Baker. Re: RFC for a new scheduling policy/class in the Linux kernel. *Linux Kernel Mailing List*, July 2009.

[7] T. Baker, M. Cirine, and M. Bertogna. EDZL scheduling analysis. *Real-Time Systems*, 40(3):264–289, 2008.

[8] T. Baker, A. Wang, and M. Stanovich. Fitting Linux device drivers into an analyzable scheduling framework. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 1–9. NICTA, July 2007.

[9] S. Baruah and G. Lipari. A multiprocessor implementation of the total bandwidth server. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*. IEEE, April 2004.

[10] 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*, pages 71–80. IEEE, August 2007.

[11] Aaron D. Block. *Adaptive Multiprocessor Real-Time Systems*. PhD thesis, University of North Carolina, 2008.

[12] 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*, pages 61–70. IEEE, July 2007.

[13] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pages 105–124. IEEE, December 2008.

[14] B. Brandenburg and J. Anderson. An implementation of the PCP, SRP, M-PCP, D-PCP, and FMLP real-time synchronization protocols in LITMUS$^{\mathrm{RT}}$. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 185–194. IEEE, August 2008.

[15] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. Manuscript. Available at http://www.cs.unc.edu/˜anderson/papers.html, May 2009.

[16] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123. Real-Time Linux Foundation, November 2007.

[17] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169. IEEE, December 2008.

[18] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE, April 2008.

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

[20] Coverity Inc. Coverity Integrity Checker. http://www.coverity.com/.

[21] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a Linux environment. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 238–245. IEEE, May 2004.

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

[23] J. Goossens, J. Anderson, and G. Fohler. Call for papers for the 22nd Euromicro Conference on Real-Time Systems. http://ecrts10.ecrts.org/, July 2010.

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

[25] M. Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, TU Dresden, 1996.

[26] S. Kato and N. Yamasaki. Global EDF-based scheduling with efficient priority promotion. In *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 197–206. IEEE, August 2008.

[27] M. Lewandowski, M. Stanovich., T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 57–68. IEEE, April 2007.

[28] J. Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM, 1995.

[29] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 410–421. IEEE, 2005.

[30] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 30:46–61, 1973.

[31] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[32] LLVM Project. Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[33] P. Mantegazza, E. Dozio, and S. Papacharalambous. RTAI: Real time application interface. *Linux Journal*, April 2000.

[34] P. McKenney. Real time vs. real fast: How to choose? In *Proceedings of the 10th Linux Symposium*, pages 57–66, July 2008.

[35] I. Molnar. Re: [PATCH][plugsched 0/28] Pluggable cpu scheduler framework. *Linux Kernel Mailing List*, November 2004.

[36] R. Pellizzoni and M. Caccamo. M-CASH: A real-time resource reclaiming algorithm for multiprocessor platforms. *Real-Time Systems*, 40(1):117–147, 2008.

[37] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, pages 476–490. ACM, January 2001.

[38] Real-Time Linux Foundation. http://www.realtimelinuxfoundation.org/.

[39] Real-Time Linux Wiki. http://rt.wiki.kernel.org/.

[40] F. Ridouard, P. Richard, and F. Cottet. Negative results for scheduling independent hard real-time tasks with self-suspensions. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 47–56. IEEE, December 2004.

[41] 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.

[42] Sparse Project. A semantic parser for C. http://www.kernel.org/pub/software/devel/sparse/.

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

[44] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pages 112–119. IEEE, June 1998.

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

[46] Y.-C. Wang and K.-J. Lin. Enhancing the real-time capability of the linux kernel. In *Proceedings of the 5th International IEEE Conference on Real-Time Computing Systems and Applications*, pages 11–20. IEEE, October 1998.

[47] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 1–12. USENIX Association, June 1993.

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