LinSched: The Linux Scheduler Simulator*

John M. Calandrino
Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599

Dan P. Baumberger, Tong Li, Jessica C. Young, and Scott Hahn
Systems Technology Lab
Intel Corporation
Hillsboro, OR 97124

Abstract

The Linux kernel 2.6.23 scheduler includes substantial changes that may entice researchers with no prior interest in Linux to attempt to understand or modify its behavior. Often, this is no easy task, particularly for someone new to Linux kernel development. Virtual machines and hardware simulators can help make the task easier; however, they introduce their own problems. Motivated by these observations, we present the LinSched *tool to host a Linux scheduler at a high simulation speed in an isolated environment, within which its behavior can be observed on a variety of different platforms* independently of other Linux subsystems. *The tool runs as a user-space process, so bugs within the scheduling code crash a single process instead of the entire system, reducing development time substantially. Experiments show that scheduling behavior within* LinSched *and the Linux kernel are nearly identical—this, combined with our personal experiences, suggest that the tool can be highly useful both for gaining an understanding of the scheduler and for the rapid prototyping of new Linux scheduling policies.*

1 Introduction

A recent surge of interest in the scheduler of the Linux kernel led to significant changes in kernel version 2.6.23. The new scheduler introduces a scheduling policy (the Completely Fair Scheduler) that attempts to provide fairness properties that may be of interest to some researchers. Additionally, the scheduler has been redesigned as a framework that organizes scheduling policies into ordered classes—when a scheduling decision needs to be made, policies are consulted in turn based on the ordering. This redesign can make it considerably easier to implement scheduling policies that subsume the default policies (classes) of the Linux scheduler, by creating a new scheduling class that is consulted prior to any of the default classes. These changes may encourage researchers who typically do not attempt kernel development to implement and empirically evaluate scheduling policies within Linux. For these researchers and other Linux developers, a good understanding of this new scheduler is crucial.

While the new Linux scheduler is well-documented, it is typically difficult to fully understand the behavior of a Linux subsystem without direct experimentation, especially if planning to modify it in further development or research. This would typically involve observing the behavior of the scheduler in a variety of synthetic test cases, followed by an attempt to change the scheduler to implement a new scheduling policy, where it is determined how those changes alter its behavior. These tasks can be tedious or frustrating when the scheduler is not yet well understood, and even sometimes when it is, for several reasons.

First, tracing the scheduler behavior can be difficult since the presence of the tracing mechanism itself may introduce bugs or change the behavior of the scheduler. In addition, test cases that stress the system (*e.g.*, four FIFO real-time tasks running on a four-processor machine, which could prevent all other tasks from executing indefinitely) may cause the system to become unresponsive or crash, making it difficult to determine the types of workloads that the scheduler can support. Second, bugs that are typically straightforward to diagnose and fix in user-space processes can be highly problematic when they occur within the scheduler. For example, an invalid memory reference in the scheduler will almost certainly crash the system and require a reboot. Debugging information is limited, and not easily obtained or understood by a new developer. This combination of long crash-reboot cycles and limited debugging information can result in a time-consuming development process. A bug that might take ten minutes to diagnose and fix in a user-space program could take hours to fix when it occurs within the Linux kernel. These issues are further complicated by the fact that changes made to the scheduler may violate implicit assumptions that are made within other Linux subsystems, which can result in deadlock or a system crash. While it is important to ultimately handle these issues, it would be preferable to debug the scheduler itself first, rather than diagnosing scheduler bugs and integration-related issues concurrently. Clearly, these issues can be problematic, especially to new Linux developers, and it would be helpful to have tools that could ease their burden. The presence of such tools may also increase

*The first author was supported by grants from Intel and IBM Corps., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425.

interest in Linux scheduler development, which could help to improve the overall quality of the scheduler itself.

Virtual machines and hardware simulators may help to ease the burdens outlined above; however, they introduce their own problems. Both tools remove the need to reboot Linux on physical hardware, but booting Linux within a virtual machine can take just as long, and booting within an architecture simulator can take several orders of magnitude longer. Some architecture simulators allow checkpoints to be taken so that the entire boot process does not have to occur every time a simulation is run. While this would assist the debugging process, a full reboot would be required whenever the scheduling code is changed (assumedly frequently). Further, tracing and debugging facilities within these environments are better than real hardware, but still limited—this is especially true for multiprocessors.

In this paper, we present a tool called LinSched, or the Linux Scheduler Simulator. LinSched hosts a Linux scheduler in an isolated environment, within which its behavior can be observed for a variety of different platforms and workloads *independently of other Linux subsystems*. The tool runs as a user-space process, so bugs within the scheduling code simply result in the termination of the LinSched process, immediately after which the scheduling code can be modified and LinSched restarted. LinSched can also be attached to the GNU Debugger (GDB) [7] and debugged with the same ease as any other user-space process. Since LinSched only simulates the scheduling subsystem, it can produce scheduler traces for a variety of workloads in considerably less time than would be required within the Linux kernel. For example, a trace of the scheduling decisions made for a large workload running on a four-processor machine over one minute of execution can be produced in about one second. Additionally, LinSched has negligible startup time as compared with tools such as virtual machines and architecture simulators.

The primary goal of LinSched is to provide a tool for observing and modifying the behavior of the Linux scheduler, and prototyping new Linux scheduling policies, in a way that may be easier or less tedious to many developers than using Linux itself. This is especially relevant considering the significant recent changes to the Linux scheduler discussed earlier. Scheduling behavior can be observed, and ideas for policies initially tested, without the need for a full implementation within Linux. This allows researchers to test many policies and select only the most promising ones to implement and test on real hardware. Since the Linux scheduler is hosted in LinSched with minimal changes, and macros are used to "ignore" or replace code rather than delete it, porting changes back into the Linux kernel is a relatively straightforward process. LinSched is general enough so that a tool such as `diff` can be used to update LinSched when new kernel versions are released, after which it can be propagated to developers for their use.

Additionally, LinSched may be of some use in developing schedulers for large-scale platforms that do not yet exist. Multicore platforms are now the standard in desktop and server systems, and the core counts of such platforms are increasing—indeed, Intel has built test chips with 80 cores and is planning on increasing the number of on-die cores in released chips over the next few years [4]. A tool such as LinSched may provide us with initial insight into the scalability issues that will arise on these platforms, with respect to developing effective scheduling policies. Virtual machines are lacking in this area, as they typically cannot support a large number of CPUs, and architecture simulators are painfully slow for large core counts.

Related work. User Mode Linux (UML) [8] is a port of the Linux kernel that runs in user space on top of another Linux kernel. A similar project, Plex86 [6], creates a virtual machine that is optimized to host a Linux guest OS. Finally, Bochs [1] is a portable open source IA-32 emulator capable of running many operating systems. All of these tools simulate the entire operating system, and are limited to a small number of CPUs. Full-system architecture simulators such as Simics [5] exist that can simulate platforms with large CPU counts, but as stated earlier, they are often slow even in their fastest run modes.

Contributions. In this paper, we present LinSched, which has the following benefits over existing tools.

- Hosts the scheduler as an isolated subsystem in user space, for easier debugging of scheduler code. This allows for the rapid prototyping and initial evaluation of new scheduling policies. Porting code back into the Linux kernel for further testing and evaluation is relatively straightforward due to a high degree of code sharing between LinSched and Linux.
- Allows scheduler behavior to be easily and quickly observed for a variety of workloads and platforms. This includes platforms that substantially differ from the platform on which LinSched is run, and workloads that severely stress the platform being simulated (and might cause a real system to freeze).
- Eliminates the need for a separate tracing mechanism that may alter scheduler behavior or introduce bugs.

Experiments presented in Sec. 3.2 show that the behaviors of LinSched and the Linux scheduler are nearly identical. An early version of LinSched has already been used to add soft real-time support to a Linux scheduler running on an asymmetric multicore platform [3]—we believe that using LinSched cut overall development time substantially, and porting the resulting code from LinSched into the Linux kernel was relatively painless. LinSched is also currently being used in an internal Intel cache-simulation project.

The rest of this paper is organized as follows. Sec. 2 discusses the design and architecture of LinSched. Sec. 3 presents an evaluation of LinSched that demonstrates its ease of use and the strong correlation of its behavior with the Linux scheduler. We conclude in Sec. 4.

2 LinSched Overview

We discuss the design of LinSched in Section 2.1 and present its high-level architecture in Section 2.2. The most recent version of LinSched will soon be available for download at http://www.cs.unc.edu/~jmc/linsched.

2.1 Design and Features

LinSched has one primary design goal: to provide a tool for easily observing and modifying the behavior of the Linux scheduler, and for prototyping new Linux scheduling policies. To achieve this goal, detailed modeling of the platform and other Linux subsystems is not required. Thus, LinSched models only the Linux scheduler and the platform topology that it requires. We now list the features of LinSched that support our main design goal.

Scheduling policy support. LinSched is able to accommodate many Linux scheduling policies. In fact, it can model virtually any policy that calls a `scheduler_tick()` function at every (periodic) timer interrupt and a `schedule()` function whenever a scheduling decision needs to be made.

Task specification. Linux allows tasks to be classified into categories, such as SCHED_FIFO (a FIFO real-time task) or SCHED_BATCH (a compute-intensive batch task). These categories, along with some notion of priority, determine both the policy (class) that is used to schedule each task, and when each task is scheduled by that policy. Creating different types of tasks with varying priorities is straightforward in LinSched. Additionally, we can specify other task characteristics, such as how long a task runs, or when it suspends (*e.g.*, to simulate I/O).

Debugging. LinSched provides a friendly debugging environment, which we achieve by hosting the Linux scheduler code in a user-space process. Thus, bugs in the code will cause only the LinSched process to terminate, and it can be easily attached to GDB for diagnosis.

Source compatibility. The scheduling code within LinSched is taken directly from the Linux scheduler with minimal modifications to remove or replace code for handling interactions with other subsystems such as memory management. Macros are used so that the resulting code looks identical to the Linux scheduler when a simulator-specific flag within the code is cleared. Thus, porting scheduling code between Linux and LinSched is relatively straightforward, and translation errors are minimized.

Data collection. Data and trace information is collected by adding code to perform bookkeeping tasks. LinSched only simulates the scheduler, instead of actually executing tasks, so this code does not alter scheduler behavior.

Automation. To facilitate rapid evaluation of multiple scheduling policies, LinSched programs can be parameterized so that many different workloads and platforms can be tested in rapid succession through the use of batch scripts.

Scalability. Future platforms will likely consist of many more processing cores than the platforms of today. Thus, LinSched is able to run larger simulations involving tens or possibly hundreds of cores relatively quickly.

Topology specification. LinSched currently supports a "flat" topology where all CPUs are in the same scheduling domain, and can be easily extended to simulate hierarchical topologies as they continue to become more commonplace.

2.2 Architecture

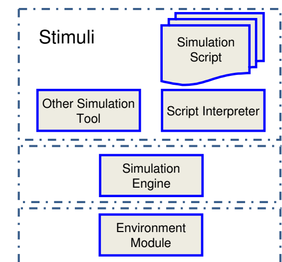

Figure 1: The LinSched architecture.

LinSched consists of three main components, as illustrated in Fig. 1. The *simulation engine* (Sec. 2.2.1) presents an API that can be used to initialize and control a simulation, and calls the appropriate scheduler functions to simulate the Linux scheduling policies. The *environment module* (Sec. 2.2.2) provides an abstraction of the Linux kernel in terms of code dependencies such as functions and macros (most of which are supplied directly by including Linux source and header files), and is responsible for presenting an appropriate platform topology to the simulation engine. Finally, *stimuli* (Sec. 2.2.3) are provided by a scripting interpreter or another tool that uses the API of the simulation engine to create tasks and start a simulation. This component can be parameterized, so that batch scripts can be used to run simulations where many different types of workloads and platforms are investigated.

2.2.1 Simulation Engine

The simulation engine provides an API that can be used by scripting tools to run simulations, and calls the appropriate functions to simulate the Linux scheduling policies. Additionally, the API methods call functions that create tasks and initialize the kernel environment, many of which are supplied by the environment module.

Much of the code within both the simulation engine and the environment module is simply a small subset of the Linux kernel source files, with modifications to provide support for certain features when necessary. A simulation-specific flag is used liberally throughout the code, and when set during compilation, the source code changes required by LinSched take place—when the flag is cleared, all source files are identical to the original Linux source code. Such a flag should make porting code between LinSched and Linux considerably easier. Thus, the task- and

runqueue-related structures that are used within the Linux scheduler are used almost verbatim within LinSched.

The API provided by the simulation engine includes the following functions, among others.

- `linsched_init()`: Initializes the scheduling subsystem and environment.
- `linsched_default_callback()`: An empty callback function. Task callbacks are performed whenever a task is scheduled, to determine task behavior or perform bookkeeping. (An empty callback simulates a task executing an infinite loop or compute-intensive code for the duration of the simulation.)
- `linsched_create_normal_task()`: Creates a "normal" task, and requires a *nice* value (indicating priority) and a task callback to be provided.
- `linsched_create_RTfifo_task()`: Creates a FIFO real-time task, and also requires both a task priority and callback.
- `linsched_change_cpu()`: Changes the CPU currently being simulated.
- `linsched_run_sim()`: Runs the simulation for a specified number of timer ticks. The `scheduler_tick()` function of the Linux scheduler is called for every CPU at each tick, and the `schedule()` function may be called to make scheduling decisions according to the currently implemented scheduling policies.

An example showing how these functions can be used to run a simulation is provided in Sec. 3.1.

The core of the simulation engine is arguably the `linsched_run_sim()` function. This function emulates the periodic timer interrupts that are generated for every CPU. These timer interrupts call the `scheduler_tick()` function, which determines if a scheduling decision needs to be made—if so, `schedule()` is called for that CPU. In LinSched, timer interrupts are emulated by a loop in which `scheduler_tick()` is called for each CPU in a random order. This models real systems more closely because the arrival order of timer interrupts is system-dependent, and in many cases, is essentially random for our purposes (*e.g.*, if interrupts depend on the order in which CPUs initialize their local (APIC) timers). Note, however, that if a more deterministic ordering is desired, then the `linsched_run_sim()` function can be easily modified to support this—for example, CPU 0 could be responsible for distributing an interprocessor interrupt (IPI) to all other CPUs, and thus may always call `scheduler_tick()` before any other CPU. We simulate the call to `scheduler_tick()` from a particular CPU by changing the processor "context" in which the function is called (with `linsched_change_cpu()`), so that calls to macros such as `smp_processor_id()` (to get the ID of the current CPU) and `current` (a pointer to the task currently scheduled on a CPU) work correctly.

Since LinSched is a user process, the scheduler can be traced by outputting data to a file or the screen. Such data output within task callbacks can easily provide us with a clear view of LinSched scheduling behavior.

2.2.2 Environment Module

The environment module provides an abstraction of the Linux kernel by satisfying the code dependencies of the simulation engine and presenting an appropriate platform topology. As stated earlier, code dependencies are typically satisfied by including a subset of the Linux kernel source files, augmented with a simulation-specific flag so that the files remain unchanged when that flag is cleared. This is particularly important when code must be changed to emulate functionality that would be difficult or impossible to support directly, *e.g.*, the `per_cpu` areas provided by Linux. In Linux, the platform topology would be acquired and provided to the scheduler during the boot process—in our case, this topology is generated during LinSched initialization based on the number of CPUs that are specified in a configuration file (though this could be parameterized for batch scripts). Currently, we support a "flat" topology where all CPUs are in the same scheduling domain—however, the environment module can be extended, and configuration file modified, to simulate other topologies. While the simulated CPUs are symmetric, it is possible to use task callbacks to simulate certain types of processor asymmetry, so that tasks perform differently on different CPUs. Such callbacks can easily reference a globally-visible data structure containing platform information.

2.2.3 Stimuli

Stimuli for the simulation are provided by a scripting interpreter or another tool that uses the API of the simulation engine to run a simulation. As LinSched does not actually execute tasks, workloads are specified in the form of task sets using the API. These tasks execute callback functions when scheduled, which determine how the tasks behave within the scheduler and environment. For example, a task callback may be implemented so that the task behaves differently on CPUs with different capabilities or cache layouts, or on a CPU on which it has been scheduled before (as it might be cache "warm" or "hot" on that CPU). An empty callback function would be equivalent to a task that executes an infinite loop or some other compute-intensive code that does not complete during the simulation. An example showing how to create a simple, non-parameterized program that provides the stimuli for a LinSched simulation is provided in Sec. 3.1. Note that stimuli could also be provided through the use of the simulation engine API within a larger simulation tool that simulates other Linux subsystems in addition to the scheduler.

3 Evaluation

In this section, we present an evaluation of LinSched, including both an example to demonstrate its ease of use, and experiments to show how well its behavior correlates with the Linux scheduler. Sec. 3.1 presents an example program that uses the simulation engine API to run a LinSched simulation, and Sec. 3.2 presents experimental results.

3.1 Running a LinSched Simulation

Fig. 2 shows a sample C program that uses the simulation engine API to create a workload and run a LinSched simulation. The workload consists of seven tasks: three "normal" tasks with a *nice* value of zero, one normal task with a *nice* value of -5, one batch task with a *nice* value of 1, one real-time FIFO task with priority 90, and one round-robin real-time task with priority 80.* Each task is created by a function with two arguments: a pointer to a callback function and the task *nice* value or priority. The function `ls_annc_cb()` is a callback that displays a message whenever the task is scheduled on a CPU. This callback allows us to generate a trace of scheduling behavior that can be parsed and analyzed—in experiments discussed in Sec. 3.2, we do exactly that when determining how well LinSched behavior correlates with the Linux scheduler.

After task creation, we run the simulation for $2 *$ `LINSCHED_TICKS` ticks. Note that it is easy to create new tasks that begin execution at some time after the simulation has begun. In this program, the round-robin task is not introduced into the system until the simulation is 50% complete, since we run the simulation for equal numbers of ticks before and after the task is created. (We could also introduce new tasks by modifying `linsched_run_sim()` directly.) Note that actual scheduling behavior is determined by the Linux functions `scheduler_tick()` and `schedule()`, among others—in this case, these functions implement the normal 2.6.23 scheduling policies.

3.2 Experimental Results

We now present the results of experiments that demonstrate how well scheduling decisions within LinSched correlate with the behavior of the Linux scheduler, and how well LinSched models scheduling behavior on larger platforms.

In our first set of experiments, we compared the scheduling behavior of Linux 2.6.23 running on an SMP machine containing four Intel Xeon 2.7 GHz processors to similarly configured LinSched simulations. The kernel was modified to allow a trace of scheduling activities to be obtained in a relatively light-weight manner (first used in [2]), and a user-space program was created to launch tasks.

We considered four different workloads in this first set of experiments. These workloads consisted of 40 tasks each,

*More information on the types of tasks described here is available by typing `man sched_setscheduler` at any Linux terminal running kernel version 2.6.16 or later.

```
   /* Include necessary header file to run LinSched simulations. */
1  #include "linsched.h"

2  int main(int argc, char **argv)
3  {
       /* Initialize simulator. */
4      linsched_init();

       /* Create 3 normal tasks with nice value 0. */
5      linsched_create_normal_task(&ls_annc_cb, 0);
6      linsched_create_normal_task(&ls_annc_cb, 0);
7      linsched_create_normal_task(&ls_annc_cb, 0);

       /* Create one normal task (nice value -5), batch task, and FIFO task. */
8      linsched_create_normal_task(&ls_annc_cb, -5);
9      linsched_create_batch_task(&ls_annc_cb, 1);
10     linsched_create_RTfifo_task(&ls_annc_cb, 90);

       /* Run simulation for LINSCHED_TICKS ticks (in linsched.h). */
11     linsched_run_sim(LINSCHED_TICKS);

       /* Create round-robin task. */
12     linsched_create_RTrr_task(&ls_annc_cb, 80);

       /* Run simulation for LINSCHED_TICKS additional ticks. */
13     linsched_run_sim(LINSCHED_TICKS);

14     return 0;
15 }
```

Figure 2: A sample LinSched simulation program.

representing the following task sets, each of which was run for one minute (simulated in LinSched).

(a) All normal tasks with a *nice* value of zero.
(b) All normal tasks—tasks 1 through 40 have *nice* values from -20 to 19, respectively. (Lower *nice* values indicate higher priority.)
(c) Half batch tasks and half normal tasks. Tasks 1-10 and 11-20 are batch tasks with *nice* values of 0 and 10, respectively. Tasks 21-30 and 31-40 are normal tasks with *nice* values of 0 and 10, respectively.
(d) Three real-time tasks, and 37 normal tasks: one FIFO real-time task with priority 90, two round-robin real-time tasks with priority 80, and 37 normal tasks with *nice* value 0. (Lower values indicate higher priority, and real-time tasks are prioritized over other tasks.)

All tasks used the `ls_annc_cb()` callback described in Sec. 3.1, thus a trace of scheduling decisions was generated. Tasks executed indefinitely (*e.g.*, in an infinite loop).

Results. The results of the first set of experiments are shown in Fig. 3, where the processor time received by each task is presented within both the Linux kernel and LinSched. In Fig. 3, insets (a) through (d) correspond to task sets (a) through (d) above. Tasks were "mapped" to identical identifiers (1 through 40) so that their behavior within Linux and LinSched can be directly compared easily.

Overall, LinSched *produces results that are highly correlated with the scheduler behavior of Linux*—external interference within the Linux kernel (*e.g.*, due to interrupts or the tracing infrastructure) accounts for the largest differences. Where the behavior of LinSched deviates from the Linux scheduler, it tends to achieve results that better approximate intended scheduler behavior. For example, in inset (a), all tasks have identical priorities, and should achieve

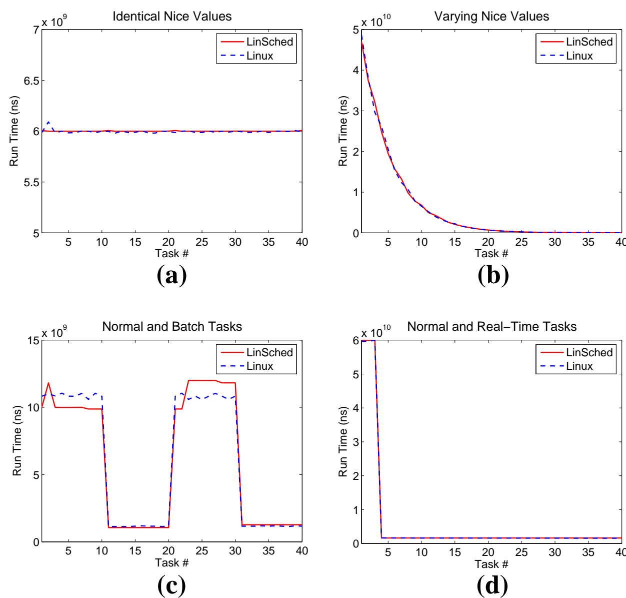

Figure 3: Processor time received by the tasks in four different workloads, in LinSched and Linux.

virtually the same execution times. In inset (c), batch tasks are supposed to receive a slight penalty as compared to normal tasks. In both cases, the expected behavior is better demonstrated in LinSched. Insets (b) and (d) indicate that LinSched behavior for these workloads is nearly identical to Linux. Note that these results are especially interesting since the time granularity within LinSched is coarser than that of Linux, as we only make scheduling decisions at each tick—in the absence of interrupts and external interference, this approach appears to be adequate for our workload.

Large-scale simulation. We next demonstrate how LinSched models scheduler behavior on larger platforms by simulating a scaled-up version of workload (b) on a 32-processor platform, again for one minute of simulated run time. This workload contains 320 tasks with *nice* values uniformly distributed between -20 and 19—groups of eight tasks share the same *nice* value. Results are shown in Fig. 4. Note the similarity between these results and those in Fig. 3(b)—such results closely approximate expected scheduler behavior. This suggests that LinSched can be used to provide initial insight into how scheduling policies will perform on larger platforms, and the scalability issues that may arise. While these results are preliminary, we believe from personal experience that LinSched is a very useful tool for exploring policies on platforms of all sizes.

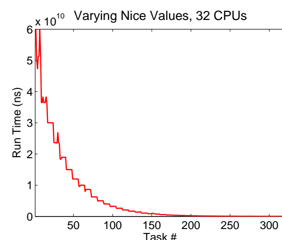

Figure 4: Results for the 32-processor workload.

4 Conclusion

Understanding and experimenting with the Linux scheduler can often be a frustrating or time-consuming process, especially for those new to Linux development. We present the LinSched tool, which hosts the Linux scheduler as an isolated subsystem in user space, for easier debugging of and experimentation with new and existing scheduling policies. This tool can be extremely useful in the early stages of scheduling policy development, as it allows for the rapid prototyping and evaluation of many different policies, from which the most promising ones can be selected for further detailed experimentation. Porting code back into Linux is made less difficult since LinSched shares most of its code with the Linux scheduler and related subsystems. In our evaluation, we demonstrated that LinSched is relatively easy to use and that it generates results that are strongly correlated with actual Linux scheduler behavior. Our personal experiences [3] using LinSched support these claims.

We want to expand upon this work in several ways. First, we would like to strengthen the automation features of LinSched to easily support experimentation with many different types of platforms and workloads. Second, we wish to improve support for hierarchical topologies, particularly those arising in NUMA, multicore, and multithreaded platforms. Finally, we are in the process of making LinSched publicly available, and will release it once that process is complete. We are very interested to see how other researchers might extend LinSched to suit their needs.

References

[1] Bochs Project. bochs: The IA-32 Emulator Project. http://bochs.sourceforge.net.

[2] 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? *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2008.

[3] John M. Calandrino, Dan Baumberger, Tong Li, Scott Hahn, and James H. Anderson. Soft real-time scheduling on performance asymmetric multicore platforms. *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2007.

[4] C. Farivar. Intel Developers Forum roundup: four cores now, 80 cores later. http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[5] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58. IEEE, 2002.

[6] Plex86 Project. The new Plex86 x86 Virtual Machine Project. http://plex86.sourceforge.net.

[7] Free Software Foundation. GDB: The GNU Project Debugger. http://sourceware.org/gdb.

[8] User Mode Linux Project. User Mode Linux. http://user-mode-linux.sourceforge.net.