Soft Real-Time Scheduling on Performance Asymmetric Multicore Platforms *

John M. Calandrino[2], Dan Baumberger[1], Tong Li[1], Scott Hahn[1], and James H. Anderson[2]
[1]Intel Corporation, Hillsboro, OR
[2]Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

This paper discusses an approach for supporting soft real-time periodic tasks in Linux on performance asymmetric multicore platforms (AMPs). Such architectures consist of a large number of processing units on one or several chips, where each processing unit is capable of executing the same instruction set at a different performance level. We discuss deficiencies of Linux in supporting periodic real-time tasks, particularly when cores are asymmetric, and how such deficiencies were overcome. We also investigate how to provide good performance for non-real-time tasks in the presence of a real-time workload. We show that this can be done by using deferrable servers to explicitly reserve a share of each core for non-real-time tasks. This allows non-real-time tasks to have priority over real-time tasks when doing so will not cause timing requirements to be violated, thus improving non-real-time response times. Experiments show that even small deferrable servers can have a dramatic impact on non-real-time task performance.

1 Introduction

In this paper, we discuss an approach for supporting soft real-time periodic tasks in Linux on performance asymmetric multicore platforms, or AMPs. Such architectures consist of a large number of processing cores on one or several chips, all capable of executing the same instruction set; however, each core may exhibit substantially different levels of performance. Performance asymmetry is in contrast to *functional asymmetry*, where each core has a different set of "capabilities" and tasks must be matched with cores possessing the capabilities they need.

AMPs may ease the transition for software developers from single-core platforms and multi-core platforms containing a few large, powerful cores to platforms containing tens or hundreds of smaller, simpler cores where exploiting parallelism will be required in order to achieve performance improvements. An example of such an architecture is shown in Fig. 1. Applications that are highly parallelizable would benefit from using a large number of the slower cores, while applications that are not easily parallelized can use the faster cores. The same level of flexibility cannot be found in a symmetric platform, assuming the same chip area for both platforms. A highly parallelizable application might perform poorly on a platform with only a few fast cores, whereas an application that cannot be easily parallelized would perform poorly on a platform with a large number of slower cores. Further arguments in favor of AMPs are given in [13], where such an architecture is called a *single-ISA heterogeneous multicore architecture*.

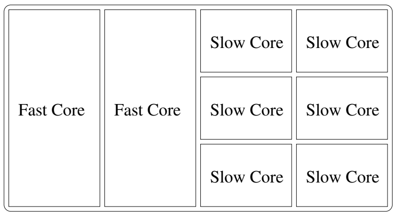

Figure 1: An AMP architecture, consisting of two larger "fast" cores and six smaller "slow" cores.

There are many cases where it may be beneficial to provide periodic soft real-time task support in a general-purpose operating system such as Linux. Such support allows the share of the system received by a given task to be controlled, which enables more predictable system and application performance. Note that, while we are mainly interested in guaranteeing a lower bound on processor shares, this support also provides an upper bound on shares, which may be useful as a form of rate control for tasks. For example, we could use this support to limit a long-running, processor-intensive task to a certain share of the machine in a more predictable way than using Linux *nice* values or priority levels.

Providing periodic soft real-time task support could be particularly beneficial to *multimedia applications*, as such applications often suffer substantial performance degradation under heavy system workloads in many general-purpose operating systems. Real-time guarantees for multimedia applications will become more important as both systems and workloads increase in complexity. This is likely to happen. Indeed, one envisioned application of multicore platforms is as *multi-purpose home appliances*, with one machine serving

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425. This research was conducted while the first author was at Intel for a summer internship.

many of the computing needs of a home. These may include:

- Multimedia applications, *e.g.*, streaming from multiple live and stored video sources, recording video for playback later, holding videoconferencing sessions, *etc*.
- Monitoring other systems in the house, such as heating/cooling, and reacting appropriately.
- Providing processing capability for one or more user terminals, each emulating the functionality of a desktop system.

Additionally, note that some multimedia applications can be very processor-intensive, *e.g.*, streaming from an HDTV video source. Such applications would require tasks with higher utilizations than are typically considered "normal" for real-time tasks. We want to provide support for real-time workloads that include such tasks.

In such an environment, support for recurrent (*e.g.*, periodic) real-time tasks is needed to guarantee the performance of real-time *and non-real-time* workloads. Less explicit approaches, such as giving real-time tasks very high static priority, are unacceptable, as they may prevent all tasks from effectively utilizing the system. We chose to provide real-time task support in Linux over other operating systems as Linux is free, open-source software that is easy to obtain and modify, and is widely accepted by both developers and end users.

Contributions. Our contributions are as follows.

- We present an approach for scheduling periodic soft real-time tasks on an AMP. This approach allows real-time tasks to enter and leave the system dynamically, and attempts to provide good performance for non-real-time applications.
- We identify deficiencies in periodic soft real-time task support in the Linux kernel and provide ways for alleviating these deficiencies.
- We experimentally determine the impact of heavy real-time workloads on non-real-time tasks in Linux, and how effectively we can minimize that impact through the use of deferrable servers (DSs) [21] of various sizes (*i.e.*, core utilizations) and background scheduling.

We implemented and evaluated our approach in both *schedsim*, a Linux scheduler simulator (discussed further in [6]), and the Linux kernel.

Note that converting Linux into a real-time operating system (RTOS) is *not* one of our goals. We provide real-time support in Linux for an end user that either prefers Linux or does not want to purchase and run a separate operating system for real-time applications. We focus on *soft* real-time support since the timing constraints in multimedia applications are usually soft. Furthermore, Linux is ill-suited for providing hard real-time support, as it is a monolithic operating system where tasks are subject to execution delays due to interrupt handling, critical sections, and system overhead. Even if the length of these delays are bounded, supporting hard real-time tasks is problematic.

We seek to maintain the major features of the Linux scheduler so that non-real-time task performance conforms as much to standard Linux as possible. To this end, we determined what deficiencies exist regarding real-time task support in Linux, and propose a minimal set of modifications required to correct these deficiencies. Our approach for correcting these deficiencies relies on the partitioned, static-priority model that is already partially supported by existing scheduling code in the Linux kernel. As a result, it requires minimal modification to the existing kernel code, and thus is more likely to result in a system that is reasonably robust and predictable for all types of tasks. Since non-real-time task performance, specifically throughput and response times, is more important to us than the size of the real-time workload that Linux can support (as the real-time workload will often be small relative to the non-real-time workload), we chose this approach rather than a global or dynamic-priority real-time scheduling approach; however, we plan to explore such approaches in future work.

To further minimize the impact of a real-time workload on the performance of non-real-time tasks, we implemented a DS on each core (we assume a single hardware thread per core) with higher priority than *any* real-time task. A DS has a *budget* that is replenished every *period*, and has a certain share of a core. When the budget is positive, non-real-time tasks are prioritized above real-time tasks, which may improve non-real-time task performance. The budget is decreased when non-real-time tasks run until it becomes zero, at which time real-time tasks are prioritized above non-real-time tasks so that real-time deadlines are met.

We evaluated the effectiveness of DSs of various sizes. We found that a DS share of up to 50% on each core provided an increasingly substantial performance benefit for non-real-time tasks while resulting in only a small decrease in the size of the real-time workload that the system can support. The partitioned approach we take for real-time task scheduling requires "bin-packing" tasks onto cores, and it is unlikely that tasks can be packed onto cores tightly enough to fully utilize every core. A small DS can often fit into the remaining (otherwise unused) share on each core. While other types of servers exist, we have chosen the DS due to its responsiveness to non-real-time tasks, its suitability to a static-priority, partitioned approach, and its overall conceptual simplicity as compared with other servers. A downside of using a DS in hard real-time systems is the *double-hit* effect—a DS could prevent any real-time task from executing until it has exhausted its budget *twice*, and this must be factored into schedulability analysis. For soft real-time systems, we can instead treat the DS as a regular real-time task and account for the double-hit effect by allowing tasks to miss their deadlines by a bounded amount (see Sec. 4.2). Therefore, the use of a more complicated server is overkill. In fact, in our experiments, by not accounting for the double hit, we were able to admit up to 30% more real-time tasks in some system configurations. To our knowledge, we are the first to note this aspect of using a DS in a system with only soft real-time tasks.

The rest of this paper is organized as follows. Sec. 2 discusses related work. Sec. 3 outlines deficiencies in Linux for supporting periodic real-time tasks on an AMP. Sec. 4 discusses how these deficiencies were corrected. Sec. 5 presents an experimental evaluation of our method, both in *schedsim* and the Linux kernel. Finally, Sec. 6 concludes and discusses several areas of future work.

2 Related Work

A substantial amount of prior work has been done on both RTOSs and the scheduling of real-time tasks on performance asymmetric platforms (though not necessarily *multicore* platforms). Such platforms are better known in the real-time community as *uniform multiprocessors*. (The previously-mentioned notion of a functionally asymmetric platform would be synonymous with a *heterogeneous multiprocessor* or *unrelated parallel machine*.) This section provides an overview of related work in both areas.

2.1 Real-Time Operating Systems

Most prior work on RTOSs has focused on either uniprocessor systems or multiprocessor systems using a partitioned approach, which is the approach we take in this paper. A recent survey of this work can be found in [18]. To our knowledge, no prior work on RTOSs has addressed the issue of scheduling on asymmetric or heterogeneous platforms. There are, however, several RTOS projects that use Linux as their foundation, and thus consider implicitly, if not directly, the possibility of a non-real-time workload on the same system.

One such effort is RTLinux [22], where real-time tasks run in a thin real-time kernel, with Linux itself running on top of this kernel as a low-priority *background task*. This strategy prevents the Linux kernel from disrupting real-time tasks (specifically due to interrupts and non-preemptable kernel code), but at the same time both restricts the ability of real-time tasks to invoke Linux kernel services (making it harder to deploy a soft real-time multimedia application, for example), and may severely impact the performance of non-real-time tasks by forcing them to be scheduled at the lowest priority at all times.

To our knowledge, multiprocessor-based RTOSs were first considered as part of work on the Spring kernel [19]. The scope of Spring extended beyond stand-alone multiprocessor systems and encompassed distributed systems comprised of several multiprocessing nodes and tasks with synchronization requirements; however, asymmetry on the same node was not considered.

In other recent work that directly targets multiprocessors, Stohr *et al.* [20] presented the RECOMS software architecture, which is a framework for running a general-purpose OS and an RTOS on the same multiprocessor machine. This framework partitions the system by placing the general-purpose OS on its own processor and preventing I/O accesses from interfering with the RTOS. RECOMS was designed as an extension to RTAI [9], which is closely related to RTLinux. The static partitioning of the system may allow non-real-time tasks to perform better with the RECOMS architecture than RTLinux, especially if the non-real-time tasks are given the fastest cores (assuming the general-purpose OS could be allocated more than one core), but is less flexible than what we propose. Additionally, the work in [20] will still suffer from the problem that real-time tasks cannot invoke Linux kernel services.

Finally, LITMUS$^{\mathrm{RT}}$ [8] is a real-time, Linux-based testbed, which was developed for empirically evaluating multiprocessor real-time scheduling algorithms. This work was limited to symmetric architectures, and considered neither asymmetry nor non-real-time task performance; however, both global and dynamic-priority approaches to real-time scheduling were implemented and evaluated in this work—we directly consider neither approach here.

2.2 Scheduling on Uniform Multiprocessors

The work in [16, 17] concerns the scheduling of *non-real-time* tasks on a uniform multiprocessor. These papers define a model for uniform multiprocessors that remains mostly intact in later work. More recently, uniform multiprocessors have become a topic of interest in the real-time community. Of all prior work on real-time scheduling on asymmetric platforms, [5] is the most relevant to our work, as it discusses a static-priority approach to scheduling on uniform multiprocessors and derives a sufficient utilization bound for that approach. (Other similar work [4, 12, 10, 11] assumes the use of a dynamic-priority, earliest-deadline-first scheduling algorithm.) As with other prior work, to our knowledge, neither the ability of tasks to enter and leave the system dynamically, nor the performance of any co-present non-real-time workload, are considered. Additionally, none of these algorithms have been implemented within the scheduler of a general-purpose operating system, where problems such as those discussed in Sec. 3 may arise.

3 Linux Real-Time Deficiencies

In this section, we discuss several reasons why the current Linux real-time task support is inadequate for supporting soft real-time periodic tasks. These deficiencies were noted after studying several references on the Linux kernel scheduler [3, 7] and the kernel source code itself (version 2.6.16).

3.1 No Support for Periodic Tasks

The standard Linux kernel provides approximately 100 static priority levels for real-time tasks. A real-time task can preempt both non-real-time tasks and lower-priority real-time tasks. There is no notion of task period, execution requirement, or deadline. As a result, there are no mechanisms to ensure that real-time deadlines are met, to prevent tasks from exceeding their specified execution requirement, or to "re-release" tasks that have new jobs to schedule. There is also no method of *admission control*—we cannot perform schedulability analysis to determine whether a task can be safely added to a particular core. By *safely*, we mean that all tasks on that core, including the added task, will meet their deadlines, or miss their deadlines by some bounded amount. As platforms become more complicated, such an explicit form of admission control becomes increasingly important, as it will be very difficult to make real-time guarantees manually.

3.2 Real-Time vs. Non-Real-Time Priorities

The standard Linux kernel always prioritizes real-time tasks over non-real-time tasks. Statically setting priorities in this manner is overkill and can negatively impact non-real-time task performance. As is usually the case with real-time jobs, we assume that *as long as a real-time job completes by its deadline, there is no additional benefit to improving the response time of that job*. Therefore, a non-real-time task should be allowed to *preempt* real-time tasks in order to improve its response time, as long as doing so does not cause reasonable deadline-miss bounds to be exceeded. While this is a well-known concept in the real-time community, no notion of it exists in the real-time support provided in Linux.

3.3 Real-Time Task Migrations

Linux does not modify the priority of real-time tasks according to their level of CPU usage or other statistics, such as sleep time or level of interactivity, as it does with non-real-time tasks. This behavior is correct, as real-time tasks are allocated a static share of the system that is independent of how the share is used. However, Linux does allow real-time tasks to be migrated during load balancing. The load-balancing algorithm is run periodically in Linux in order to keep the run queues on each core at approximately the same length. On asymmetric platforms, this algorithm can be modified to maintain run-queue lengths that are proportional to the processing power of each core (*e.g.*, if core A is twice as fast as core B, then its queue should be twice as long). In either case, while this algorithm is suitable for and beneficial to non-real-time tasks, it is inappropriate for real-time tasks. This is because load is balanced on each core with respect to the *number* of tasks per core rather than the amount by which each task *utilizes* that core. A real-time task specifies its system utilization requirements up front, and (in the partitioned approach) should be placed onto a core where these requirements can be satisfied for its lifetime. Migrating such a task can affect both its execution requirement (as migrations are not free) and result in unbounded tardiness since the task is no longer on the core where its required share of the system was guaranteed. (We could perform admission control during load-balancing, but this could be costly and contrary to our rules for placing real-time tasks onto cores, stated later.) On the other hand, it is perfectly acceptable, and perhaps even preferable, to "count" real-time tasks in the task queues when performing load-balancing for non-real-time tasks, so long as real-time tasks are not migrated.

3.4 Low Timer Frequency

The default timer frequency for version 2.6.16 of the Linux kernel is 250 ticks per second. This is equivalent to a four millisecond *jiffy*, or the time between timer ticks, which represents the smallest quantum size allowable. This jiffy is too long; a jiffy of one millisecond, or 1,000 ticks per second, would provide improved timing granularity. Fortunately, the developers of Linux recognized this, and allowed the frequency of timer interrupts to be raised to 1,000 ticks per second via a configuration option when kernel responsiveness is crucially important (as is the case with real-time tasks). Thus, as a solution already exists for this issue, we simply incorporate it into our kernel, and do not discuss it further in this paper. We do note, however, that Linux does not officially support timer frequencies higher than 1,000—attempting to increase the timer frequency above 1,000 has the potential to adversely affect both non-real-time task performance (due to excessive interrupt processing overhead) and system robustness, and can "break" parts of the kernel. To safely increase the timer frequency above 1,000 ticks per second, a thorough study of what would need to change in order to support higher frequencies would be required.

3.5 Interrupts and Critical Sections

There are also timing issues related to interrupt handling and non-preemptable critical sections (due to the possession of a lock or entry into certain pieces of kernel code), particularly when an interrupt handler or critical section is long. For a soft real-time system, we would like to limit the impact of these issues on system timing so that deadline misses can be bounded. These issues have been widely noted (for the first time, to our knowledge, in [22]), and we believe that adequate solutions now exist for *soft* real-time systems, which do not require the "dual-kernel" approach used in [22], by effectively using one or more patches such as those cited in [1, 2]. Kernel spinlocks pose an additional problem, since the order in which waiting threads claim a spinlock is unpredictable, and therefore it is impossible to make timing predictions for threads using these locks; however, this problem can be easily fixed by using queue locks. Thus, at least partial solutions to these issues already exist, and as a result, we do not discuss them further in this paper.

4 Correcting Deficiencies

We now discuss how we changed the Linux kernel to correct the remaining deficiencies outlined in Sec. 3.

4.1 Adding Periodic Task Support

The modifications described in this section are the most substantial and important changes we made, as they embody the foundation of the real-time support we added. These changes included the following.

- We modified scheduling structures to include the necessary periodic task support. For example, we added variables to the task structure to specify a task's execution requirement, period, and deadline.
- We modified and added a number of scheduling functions. The tick-handling function, *scheduler_tick*, was the most modified, followed by *schedule*, the function that decides which task to run on a core.
- We provided several new system calls: one to *promote* a non-real-time Linux task* to a periodic real-time task with a specified execution requirement and period, one to *demote* such a task back to non-real-time status, and one to change the size of the DS on a core (discussed further in Sec. 4.2).

Note that these changes allow real-time tasks to enter and leave the system *dynamically*, *i.e.*, the set of real-time tasks does not have to be static. This is an important feature in a general-purpose operating system, especially when supporting multimedia workloads that may start and end at any time.

Our unit of time for specifying task periods and execution requirements is *jiffies*, as defined in Sec. 3.4. The execution requirement of a task is specified as its worst-case execution time on the *slowest core*—its execution time will be shorter on faster cores. For example, if the slowest core performs one unit of work per tick, then a core with twice that speed will perform two units of work per tick, completing a given job in half the time. (This model is very similar to that in much of the related work cited in Sec. 2.2.) We assume that all real-time tasks are independent, and therefore we do not consider issues such as data sharing between two tasks running on cores of different speeds.

A real-time task is placed in the *active queue* while it is waiting to run. This queue contains tasks that are ready for execution, and is part of the *Linux runqueue*, a complex per-CPU Linux data structure that maintains scheduling information for a logical CPU, such as the tasks currently assigned to it. (Recall that there is only one hardware thread per core, so each core maps to a single logical CPU in Linux.) Once the task is selected to execute, it runs its current job to completion or until it exceeds its execution requirement. In both cases, the task is then placed into a *real-time wait queue* until its next job release (*i.e.*, the start of its next period), at which time the task is removed from the real-time wait queue and placed back into the active queue. Real-time tasks are scheduled on each core using rate-monotonic scheduling [15].

Admission control. During task promotion, a task is placed onto the *slowest core* that can satisfy its real-time requirements. We use time-demand analysis for static-priority tasks [14], assuming rate-monotonic scheduling for periodic (non-DS) tasks, to determine whether a task can be safely added to a core, traversing the cores from slowest to fastest until a suitable core is found. If a suitable core is found, the task is added to that core; otherwise, the task cannot be promoted. (However, it may continue to run as a non-real-time task.) While not a particularly efficient algorithm in terms of speed, its cost is incurred only once during the lifetime of a periodic real-time task, and it is incurred "up front" before such a task becomes part of the real-time workload. Time-demand analysis is a good sufficient test for the asynchronous periodic task model when using static-priority, partitioned scheduling approaches. (In the *synchronous* model, the test is also necessary, but since jobs can enter and leave the system dynamically, our model is by definition asynchronous.) While partitioning approaches often do not efficiently utilize the system, we are willing to accept this in exchange for the relative simplicity of our implementation, which results in increased kernel robustness and predictability, and good performance for non-real-time tasks.

4.2 Modifying Non-Real-Time Task Priorities

We allow non-real-time tasks to have priority over real-time tasks when appropriate through the use of DSs, as discussed in Sec. 1. Note that the DS *always* has higher priority than any real-time task regardless of its period. This modification can improve the response times of non-real-time tasks in the average case. We added variables representing DS periods, budgets, and other attributes to the *Linux runqueue* structure. As this is a per-CPU data structure, the budget and period of the DS can be different on every core, which may either encourage or discourage tasks to run on certain cores. As with real-time tasks, we specify the DS period in jiffies, and specify its budget with respect to the slowest core in the system. We modified the *schedule* function, which typically searches for real-time tasks to execute before searching for non-real-time tasks, to first search for *non-real-time tasks* to execute when the DS budget is positive, thus giving such tasks higher priority at that time. As mentioned in Sec. 4.1, we also added a system call that allows the size of the DS to be changed, allowing the system to better react to fluctuating workloads.

*Note that a *Linux task* may represent either a process or thread, depending on whether it shares a common address space with another task.

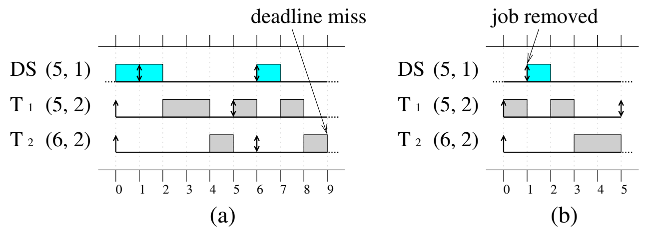

Figure 2: A DS **(a)** with an extra job due to the double hit; and **(b)** with the extra job removed. Removing the extra job allows all tasks to meet their deadlines.

Admission control with a DS. A DS, like a real-time task, must be accounted for during admission control. We treat a DS as a real-time task with priority higher than any "real" real-time task (regardless of the DS period) and an execution requirement equal to its budget. Note that we avoid accounting for the double-hit effect (discussed in Sec. 1) during admission control by allowing tasks to miss their deadlines by a bounded amount. Since deadlines may be missed, we allow the specification of a tardiness bound during admission control when adding a task—a suitable core must be able to guarantee the specified bound for all tasks, calculated as explained in Theorem 1 below. In this theorem, e_s and p_s denote, respectively, the budget and period of the DS, and e_i and p_i the execution requirement and period of a task T_i.

Theorem 1 *Consider a task system τ consisting of a DS and n tasks $T_1, \ldots, T_n$ in order of increasing periods, where the DS is given the highest priority, and the remaining tasks are prioritized against each other in rate-monotonic order. If τ is schedulable with the DS treated like an ordinary real-time task* (i.e.*, ignoring the double-hit effect*)*, then task T_i's tardiness is at most $max(0,\ (min\ t : t \geq \lceil \frac{t}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k + (e_i + e_s)) - p_i)$.*

Proof: Because τ is schedulable when the DS is treated like an ordinary task, any tardiness is due to the double-hit effect. Consider any job J of task T_i and let t_d be its deadline. Because of the double-hit effect, up to e_s time units of work due to jobs of T_i released prior to t_d may remain unfinished at t_d. However, *no more than e_s time units of such work can remain unfinished*; otherwise, J would miss its deadline in the absence of the double-hit effect, which is a contradiction. To see this, suppose that we remove the extra job of the DS in the double-hit scenario, as illustrated in Fig. 2, and allow commensurate portions of jobs of $T_1, \ldots, T_i$ to shift forward. If more than e_s time units of work of T_i were pending at t_d, then after shifting, some of the computation for J would still be scheduled after t_d. This argument can be generalized to show that, at any time, the *total* amount of unfinished work due to jobs that have missed their deadlines is at most e_s.

It follows that we can determine an upper bound on the response time of *any* job J of T_i by using ordinary time-demand analysis, adjusted to account for up to e_s time units of additional demand, which is due to the DS or earlier jobs that have missed their deadlines. This additional demand can be accounted for by treating J as if its execution cost were $(e_i + e_s)$ instead of e_i. In this case, J's response time is $(min\ t : t \geq \lceil \frac{t}{p_s} \rceil \cdot e_s + \sum_{k=1}^{i-1} \lceil \frac{t}{p_k} \rceil \cdot e_k + (e_i + e_s))$. If this value exceeds p_i, then subtracting p_i from it gives an upper bound on tardiness. ■

If we required *hard* real-time guarantees, we would have no choice but to consider the double-hit effect during admission control. Since we only require *soft* real-time guarantees, bounded deadline misses are tolerable, and the use of a more complicated server would be overkill. It would be interesting to investigate how other servers compare to the DS for soft real-time systems as future work.

Background scheduling. As in standard Linux, we also allow non-real-time tasks to be background scheduled whenever there is no demand from the real-time workload, even if the DS budget is not positive. Since real-time tasks are added to the slowest core that allows their timing requirements to be met, this should leave many of the faster cores either partially or fully available for non-real-time tasks, thus further improving non-real-time performance.

4.3 Preventing Real-Time Task Migrations

When a task is promoted (using a system call) and assigned to a core, we modify the processor affinity of that task so that it may *only* run on that core. This is accomplished by modifying the *cpus_allowed* bit mask of the task. This prevents such tasks from missing deadlines or being otherwise severely disrupted by being migrated unexpectedly by the load-balancing algorithm. The old bit mask, indicating the processor affinity of the task before promotion, is saved—it is used to reset the affinity of the task when the task is demoted (to non-real-time status). Note that we do not modify the processor affinity of non-real-time tasks in any way, and the load-balancing algorithm behaves as usual for such tasks.

5 Evaluation

In this section, we first present results from experiments in the *schedsim* scheduler simulator, and then provide results from experiments where we implement our approach in the Linux kernel itself.

5.1 Evaluation in *schedsim*

We created the *schedsim* simulator and used it to both implement and evaluate our algorithm. In *schedsim*, a slightly modified version of the Linux scheduling code can be simulated on some number of cores, each running at some specified speed. Tasks are allowed to enter and leave the system dynamically. This simulator allowed us to test scheduling policies in the Linux kernel on a variety of AMPs, most notably large-scale platforms that would be difficult to emulate using a real machine (as such large platforms do not exist yet). Additionally, *schedsim* allowed us to determine the effectiveness of our approach under "ideal" conditions, and to more easily obtain certain types of measurements, such as those related to deadline misses. Finally, by using *schedsim*, we were able to develop our algorithm considerably faster than if we had used Linux from the start.

Task properties. In our first set of experiments, we added 100 random real-time and non-real-time tasks (each) over time to different AMPs in *schedsim*, given a variety of different DS utilizations ranging from zero to full core utilization. The amount of time between adding new tasks was a random number of timer ticks between one and 50, resulting in an average simulation run time of about 5,000 timer ticks, or five (simulated) seconds. Real-time tasks were randomly generated such that periods varied from two to 80 timer ticks, and execution requirements varied from one timer tick to twice the task period (which required the task to be placed on a core at least twice as powerful as the slowest core in order to ensure that the task met its deadlines). We allowed tardiness to be arbitrarily large during admission control. Real-time tasks that were admitted into the system ran for the entire simulation lifetime. Note that attempting to add such a large number of real-time tasks to a system resulted in many rejected tasks—the goal was to stress the system to determine the upper limit on the real-time workload that could be accommodated.

For each non-real-time task, both a *lifetime* and *run-ratio* were specified. The lifetime of a task is the number of ticks that it runs until it completes and exits the system. The run-ratio of a task specifies the percentage of time that the task should be running as opposed to "blocking" (*i.e.*, waiting for I/O, *etc*.). At the end of each timer tick in the simulator, a random number x between zero and one was generated for each currently running (non-real-time) task. If x was less than the run-ratio of the running task, the task continued to contend for processing time; otherwise, the task "blocked." We simulated "blocking" by suspending the execution of the current task and removing it from the Linux runqueue. We then used the same random check during successive ticks to determine when to return the task to the active queue. In this way, the run-ratio specified the approximate processing demand placed by the task on the system. These run-ratios ranged between 1% and 100% for each non-real-time task, with a greater proportion of the tasks assigned lower run-ratios.

Simulated platforms. All simulated platforms were uniform memory architectures (UMAs). The fast cores in such platforms had a 3.0 GHz clock speed and the slow cores ran at 1.5 GHz, though what is more important is the *ratio* between the speeds of the fast and slow cores. In all experiments, we assumed that clock speed was the *sole indicator* of the processing power of a core (rather than a combination of clock speed, processor design, memory system performance, and other system characteristics). Thus, a fast core was exactly twice as powerful as a slow core, even if in reality this would not be the case.

We considered four system configurations, three of which had a ratio of three slow cores to one fast core (a ratio we have found provides the best overall system performance). The SMALL configuration consisted of eight cores (6 slow, 2 fast). The MEDIUM and LARGE configurations consisted of 16 (12 slow, 4 fast) and 32 (24 slow, 8 fast) cores, respectively. Finally, the EXTREME configuration consisted of eight cores, which ran at speeds 8, 8, 7, 6, 5, 3, 2, and 1 times the slow core speed (1.5 GHz), respectively. This configuration resulted in unrealistic clock speeds (*i.e.*, over 10 GHz). However, as mentioned earlier, only the performance *ratio* between cores was important in the simulation. This configuration represented "extreme" cases where asymmetry of this sort might be encountered. Some examples include cores that independently lower their frequencies due to inactivity or overheating, or a badly manufactured chip where all cores cannot run at the maximum clock speed, but could still be used at lower clock speeds.

Experimental results. We measured both the percentage of real-time tasks that were admitted into the system (and therefore only missed deadlines in the simulator due to the DS double-hit effect) and the *percentage of non-real-time task run time* (%NRT-RT). If run_time is the amount of time that a task ran, and $ready\_time$ is the amount of time that the task was ready to run, but not actually running (*i.e.*, waiting to run), then %NRT-RT would be calculated as $\frac{run\_time}{run\_time+ready\_time} \cdot 100$. Time spent "blocking" is not included in the calculation as a task does not need a core during that time. A higher percentage of time spent running as opposed to waiting to run implies increased task throughput and better response times. Results for all configurations are shown in Fig. 3, where each data point represents an average of 500 experimental runs. The DS size is specified in terms of the percentage utilization of the slowest core—the budget is calculated for a core of a given speed assuming a period of 10. In the "MAX" case, the budget is specified to fully utilize each core.

The addition of a DS substantially improves the percentage of time that non-real-time tasks run (versus waiting to run), thus implying improved responsiveness. In the SMALL and MEDIUM configurations, the improvement is most dramatic—%NRT-RT increases substantially with each 10% increase in DS utilization, while the real-time workload supported by the system decreases by only 1-2% each time. As the size of the DS continues to increase, the non-real-time performance benefit levels off.

The performance benefit decreases as the number of cores increases. This is because there is less of a need for a DS to prevent the system from being monopolized by real-time tasks since the relative system workload is smaller (the work-

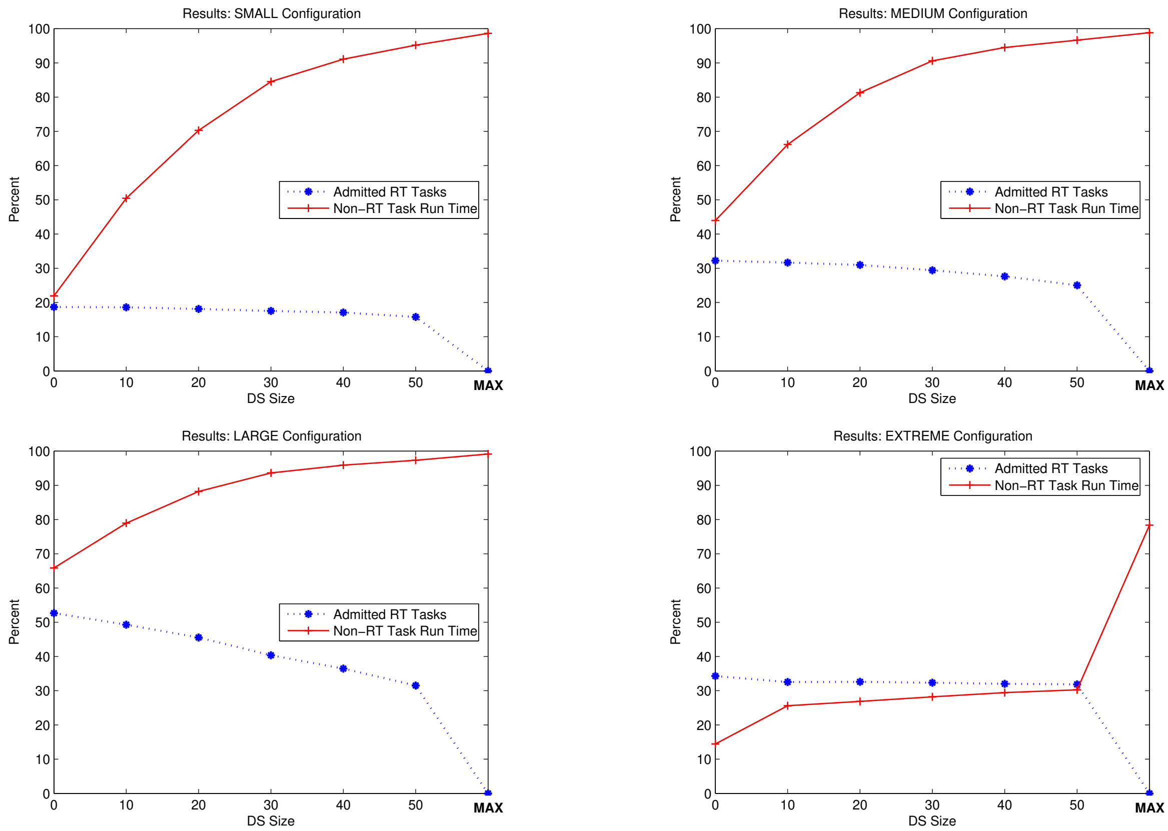

Figure 3: Percentage of admitted real-time tasks and %NRT-RT, for each configuration. Note that, for both metrics, higher numbers indicate better performance.

load is the same size for every configuration). Though a greater number of real-time tasks are also admitted onto systems with more cores, these cores are unlikely to be fully utilized, resulting in an increase in unused processing capacity and improved non-real-time performance. The impact on real-time tasks also becomes more prominent, as we are reducing the effective utilization of more cores when adding DSs. Regardless, a substantial performance benefit still exists for all system configurations when using our method.

In the EXTREME configuration, we see a benefit from adding a small DS; however, we see very little improvement from increasing its size, suggesting that as the asymmetry of the platform increases, the benefits of using a DS may decrease. This may be due to the fact that the DS size is specified with respect to the slowest core, and therefore may remain quite small relative to the faster cores in the system. Even in this system configuration, however, we still see a performance benefit.

One explanation for the benefit we see relates to the "bin-packing" of tasks onto cores required by the partitioned approach to scheduling real-time tasks. It is quite unlikely that each core can be fully utilized by the real-time workload, especially when using static-priority scheduling. The unutilized portion of a core can easily reach 20% or more of its utilization—in some cases, we may only be able to utilize roughly *half* of a core. As a result, it is often the case that a DS can fit on a core without needing to reduce the real-time workload on that core. Of course, the probability of this occurring is less likely as the size of the DS increases.

Note also that avoiding accounting for the double-hit effect during admission control allows a larger real-time workload to be placed on each core. To demonstrate this, we measured the average percentage of real-time tasks that were admitted onto each system when we re-ran the above experiments, this time accounting for the double hit during admission control. By not accounting for the double hit, we were able to admit up to 30% more real-time tasks for the largest DSs and system configurations.

We next determined by how much tasks missed their deadlines in these simulations as a result of the double-hit effect. Table 1 shows these amounts. Note that when tasks miss their deadlines, it is often by much less than the worst-case upper bound—in fact, the *median* amount by which a task missed a deadline was one tick for all configurations.

System	Avg.	Max.
SMALL	1.28	9
MEDIUM	1.26	12
LARGE	1.22	6
EXTREME	2.22	36

Table 1: Amounts by which tasks missed their deadlines (in timer ticks).

Results with lower task utilizations. Finally, we re-ran the same experiments with a maximum per-task execution

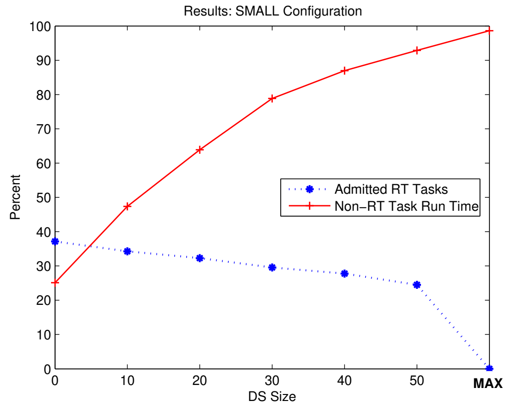

Figure 4: Results with lower task utilizations.

requirement of one half of the task period (meaning that the task would require half of the utilization of the slowest core). Results are shown in Fig. 4 for the SMALL configuration only, due to space constraints. Note that decreasing task utilizations results in a slightly steeper drop in the real-time workload that we can support as the DS size increases; however, there is still a clear benefit.

5.2 Evaluation in the Linux Kernel

We now discuss the results of experiments performed in the Linux 2.6.16 kernel. We modified the kernel to provide both soft real-time and *non-real-time* task support for AMPs. Non-real-time task support included load-balancing such that run-queue lengths were maintained that were proportional to the processing power of each core, and pushing tasks from slow cores to faster cores when possible. When DSs were used, one was placed on each core with a budget of one and a period of four, resulting in 25% of each core being reserved for non-real-time tasks, regardless of core speed. (This is different from the *schedsim* evaluation, where the size of the DS was not scaled by the core speed.)

All experiments were performed on an Intel Harwich machine consisting of an UMA with four dual-core Intel Xeon processors running at 2.6 GHz. Each core has a 1024K private L2 cache. Hyperthreading (HT) for each core was disabled. The machine has 8 GB of main memory. The AMPs of interest to us were emulated by modulating the clock speed of each processor, which we used to create three different system configurations. The SYMMETRIC configuration consisted of eight full-speed (2.6 GHz) cores. The SMALL configuration consisted of two full-speed cores and six cores modulated to 50% of their full speed. Finally, the EXTREME configuration consisted of two full-speed cores and six cores modulated to 87.5%, 75%, 62.5%, 37.5%, 25%, and 12.5% of their full speed. Note that the SMALL and EXTREME configurations are highly similar to their counterparts in the *schedsim* evaluation.

We measured the response times of non-real-time tasks that execute *busy-sleep* loops. The pseudo-code of a busy-sleep loop thread is shown in Fig. 5. During each iteration,

```
BUSYSLEEP ()
i := 0;
while i < 20000 do
    sleep_ms := RANDOM(5, 25);
    start_time := GETTIME();
    SLEEP(sleep_ms);
    end_time := GETTIME();
    response_time(i) := end_time − start_time − sleep_ms;
    i := i + 1
od
```

Figure 5: Pseudo-code of a busy-sleep loop thread.

each thread reads the current time, then sleeps for a random amount of time between 5 and 25 ms, and then reads the time again as soon as possible after awakening. The time that is read after awakening indicates the first time that the thread is able to run. The difference between the pre- and post-sleep times read, minus the sleep time itself, indicates the *response time*, or the time that the thread had to wait after awakening before it was executed. Eight threads, or one per core, running 20,000 iterations of this loop executed concurrently in the presence of a real-time workload. Response times for each iteration were measured for all system configurations, both with and without DSs.

The real-time workload consisted of eight tasks, one running on each core. Such tasks were designed to use 75% of the processing capacity of the core on which they ran. It was observed that all real-time tasks received approximately the expected percentage of CPU time, with very few unexpected deadline misses, both with and without DSs. Additionally, such tasks *maintained* the correct CPU shares when left running for several days or more. Most deadline misses occurred almost immediately after task promotion, probably due to the time required to migrate promoted tasks to the appropriate cores and make new scheduling decisions based on task priority changes. These misses may possibly be avoided by extending the deadline of the first job of each promoted task, in order to grant the system more time to stabilize before real-time constraints must be ensured. (Note that tasks may still miss deadlines, due both to ignoring the double-hit effect of the DS and sources of latency in Linux during admission control.)

Table 2 shows the results of these experiments for each system configuration. Response times were improved with DSs for all system configurations. In particular, both maximum response times (especially for the EXTREME configuration) and average response times decreased. The SYMMETRIC configuration demonstrated some of the most consistent improvements, with the average and median response times dropping significantly, and the variation in response times substantially reduced (note the reduction in standard deviation). Note that large *variations* in response times could be just as frustrating to a user of an interactive application as large response times. The improvements overall were less consistent, but still significant, in the SMALL and EXTREME configurations.

Configuration	With DSs Min.	Avg.	Med.	Std. Dev.	Max.
SMALL	6	1681	1084	1906	20899
SYMMETRIC	6	654	526	600	17044
EXTREME	6	11374	12058	8059	25059

Configuration	Without DSs Min.	Avg.	Med.	Std. Dev.	Max.
SMALL	6	1819	1078	2193	25035
SYMMETRIC	6	1554	674	2218	22017
EXTREME	8	14831	15378	8204	57894

Table 2: Response times, in μs, of busy-sleep loop iterations running alongside a heavy real-time workload, for each configuration.

6 Conclusion

We have presented an approach for supporting soft real-time periodic tasks in Linux on AMPs. We outlined the deficiencies that currently exist in Linux with respect to supporting periodic real-time tasks on such platforms and how they were alleviated. We then implemented our new method in both the *schedsim* scheduler simulator and the Linux kernel. Our evaluation showed the effectiveness of our system in both supporting real-time workloads and providing reasonable response times for non-real-time tasks.

Many future research directions were uncovered through this work. First, we would like to explore the use of global and dynamic-priority approaches to real-time scheduling on asymmetric platforms, as such algorithms might increase the size of the real-time workload that we can support. Second, we want to explore other server types besides the DS, especially in light of our claim that more complicated server types might be overkill for soft real-time systems. Third, we want to explore the possibility of a server that dynamically adjusts its size based on the current real-time workload (as opposed to allowing manual adjustment of the server size with a system call), so that non-real-time tasks always get the best response times possible. Fourth, we would like to investigate the potential for supporting timer frequencies above 1,000. Fifth, we would like to explore issues related to simultaneous multithreading (*i.e.*, hyperthreading). Sixth, we would like to explore support for soft real-time workloads on *functionally asymmetric* multicore platforms, and platforms with other types of asymmetry, *e.g.*, cache layout asymmetry. Finally, it would be interesting to consider *synchronization* issues. For example, when real-time tasks share data, admitting such tasks onto the slowest core that can satisfy their requirements may not be the most beneficial strategy—other task assignment strategies, such as assigning tasks that share data to same-speed cores, may exhibit better performance.

References

[1] Realtime and interrupt latency. http://lwn.net/Articles/139784/, 2005.

[2] Linux Real Time Patch Review - Vanilla vs. RT patch comparison. http://www.captain.at/howto-linux-real-time-patch.php, 2006.

[3] J. Aas. Understanding the Linux 2.6.8.1 CPU scheduler. Silicon Graphics, Inc., 2005.

[4] S. Baruah. Scheduling periodic tasks on uniform multiprocessors. *Information Processing Letters*, 80(2):97–104, 2001.

[5] S. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, 2003.

[6] D. Baumberger, T. Li, J. Young, S. Hahn, and J. Calandrino. *schedsim*: The many-core scheduler simulator. In submission.

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

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

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

[10] S. Funk and S. Baruah. Characteristics of EDF schedulability on uniform multiprocessors. *Proc. of the EuroMicro Conference on Real-Time Systems*, 2003.

[11] S. Funk and S. Baruah. Task assignment on uniform heterogeneous multiprocessors. *Proc. of the EuroMicro Conference on Real-Time Systems*, 2005.

[12] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. *Proc. of the 22nd IEEE Real-time Systems Symposium*, 2001.

[13] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *Proc. of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.

[14] J. Lehoczky, L. Sha, J. Strosnider, and H. Tokuda. Fixed priority scheduling theory for hard real-time systems. *Foundations of Real-Time Computing, Scheduling, and Resource Management*, 1991.

[15] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.

[16] J. Liu and C. Liu. Bounds on scheduling algorithms for heterogeneous computing platforms. *Proc. of the IFIP Congress*, 30:483–485, 1974.

[17] J. Liu and T. Yang. Optimal scheduling of independent tasks on heterogeneous computing systems. *Proc. of the ACM National Conference*, 1:38–45, 1974.

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

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

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

[21] J. Strosnider, J. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

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