On the Implementation of Global Real-Time Schedulers*

Björn B. Brandenburg and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

An empirical study of implementation tradeoffs (choice of ready queue implementation, quantum-driven vs. event-driven scheduling, and interrupt handling strategy) affecting global real-time schedulers, and in particular global EDF*, is presented. This study, conducted using UNC's Linux-based LITMUS^{RT} on Sun's Niagara platform, suggests that implementation tradeoffs can impact schedulability as profoundly as scheduling-theoretic tradeoffs. For most of the considered workloads, implementation scalability proved to not be a key limitation of global* EDF *on the considered platform. Further, a combination of a parallel heap, event-driven scheduling, and dedicated interrupt handling performed best for most workloads.*

1 Introduction

The advent of multicore systems has resulted in renewed interest in real-time multiprocessor scheduling algorithms. In work on this topic, scheduling-theoretic issues have received the greatest attention. By comparison, only little attention has been devoted to the *actual implementation* of such algorithms within real OSs on real hardware, and consequently, only little is known about how implementation tradeoffs impact schedulability. This is surprising, as it is well established that such tradeoffs play a crucial role in real-time performance on uniprocessors [16].

In light of recent algorithmic research, a particularly relevant case in point is global scheduling algorithms, which use a single shared ready queue. Very little guidance can be found in the literature concerning how to *best* implement such algorithms. Should ready queues be implemented as lists or heaps? Should sequential queues with coarse-grained locking or parallel data structures be used? Should the scheduler rely on periodic timer ticks or follow an event-driven approach? On the surface, it may seem that any of these choices are viable. However, is this really true?

In this paper, we present an evaluation of these and other implementation tradeoffs as they arise in the implementation of a *global earliest-deadline-first* (G-EDF) scheduler. We show that different implementation choices—all seemingly plausible "in theory"—can have dramatically different effects on real-time schedulability.

Prior work. This is the third in a series of papers by our group investigating fundamental questions concerning the viability of supporting sporadic real-time workloads on SMP and multicore platforms *under consideration of real-world overheads*. To facilitate this line of research, our research group developed a Linux extension called LITMUS^{RT} (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which allows different (multiprocessor) scheduling algorithms to be implemented as plugin components [9, 13]. To the best of our knowledge, LITMUS^{RT} is the only (published) real-time OS in which global real-time schedulers are supported.

In the initial study [9], Calandrino *et al.* evaluated several well-known multiprocessor real-time scheduling algorithms on a four-processor 2.7 GHz Intel Xeon SMP (not multicore) platform. On this platform with *few and fast processors*, *relatively large, private L2 caches*, and *fast memory*, each tested algorithm was found to be a viable choice in some of the tested scenarios, and global algorithms excelled at supporting soft real-time workloads.

In the second study [7], Brandenburg *et al.* explored the relative *scalability* of different real-time schedulers as implemented in LITMUS^{RT}. To do so, they ported LITMUS^{RT} to a radically different, and much larger, multicore platform: a Sun Niagara with 32 logical processors, each with an effective speed well below 1 GHz.[1] On this platform with *many slow processors*, *a relatively small, shared L2 cache*, and *slower memory*, a decrease in the competitiveness of G-EDF was noted, especially in the presence of many tasks. Clearly, this study had uncovered scalability limitations *in that particular implementation of* G-EDF, but does this imply that G-EDF-like algorithms are a "lost cause" on large multicore platforms? Could the G-EDF plugin be significantly improved, and, in terms of schedulability, would overhead reductions even matter?

Contributions. Both preceding studies [7, 9] considered several scheduling algorithms, but only one implementation per algorithm. In stark contrast, the study presented in this paper considers only one scheduling algorithm, G-

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

[1]Eight 1.2 GHz cores with four hardware threads per core. A core's cycles are distributed among its hardware threads in a round-robin manner.

1052-8725/09 $26.00 © 2009 IEEE
DOI 10.1109/RTSS.2009.23

IEEE computer society

G-EDF, but twelve possible realizations of it, seven of which were implemented and evaluated in LITMUS$^{\mathrm{RT}}$. The objective of this study was to determine which of these implementations are viable. Our major findings are as follows: **(i)** implementation tradeoffs in global real-time schedulers such as G-EDF affect schedulability *significantly*; **(ii)** there is a "best" way to implement G-EDF that outperforms other approaches in most cases; **(iii)** on our Niagara, which is a large multicore platform by today's standards, implementation scalability is *not* a key limitation of G-EDF.

In the sections that follow, we provide needed background (Sec. 2), describe the various implementation alternatives that we considered (Sec. 3), present our study and findings (Sec. 4), and conclude (Sec. 5).

2 Background

We consider the problem of scheduling n independent[2] real-time tasks $T_1, \ldots, T_n$ on m identical processors $P_1, \ldots, P_m$. LITMUS$^{\mathrm{RT}}$ supports *sporadic tasks* with *implicit deadlines*,[3] wherein each task T_i is specified by its *worst-case execution time* e_i and its *period* p_i. The j^{th} job of T_i, denoted T_i^j and released at r_i^j (where $r_i^j \geq r_i^{j-1} + p_i$), should complete by its *deadline* $r_i^j + p_i$, otherwise it is *tardy*. Note that T_i^j being tardy does not alter r_i^{j+1}, but T_i^{j+1} cannot execute until T_i^j completes (tasks are sequential). T_i's *utilization* u_i is given by e_i/p_i; the sum $\sum_{i=1}^{n} u_i \leq m$ denotes the system's *total utilization*. A job is *pending* after its release until it completes.

To avoid confusion, we use the term "task" exclusively to refer to sporadic tasks, and use the term "process" when discussing OS implementation issues. In LITMUS$^{\mathrm{RT}}$, sporadic tasks are implemented as processes, and jobs are an accounting abstraction managed by the kernel.

Scheduling. A *hard* real-time (HRT) system is considered to be *schedulable* iff it can be shown that no job is ever tardy. A *soft* real-time (SRT) system is considered (in this paper) to be *schedulable* iff it can be shown that tardiness is bounded.

We investigate G-EDF as a representative of the class of global, preemptive, priority-driven, work-conserving scheduling policies, *i.e.*, all processors use a single shared *ready queue* sorted by non-decreasing deadlines and jobs (but not necessarily the OS!) can be preempted.

There are two fundamental ways to realize schedulers (see [10, 16] for overviews):

S1 *event-driven scheduling*, wherein a job is scheduled *immediately* when there are fewer than m higher-priority[4] jobs pending (either upon release or when a higher-priority job completes);

S2 *quantum-driven scheduling*, wherein real-time jobs are only scheduled at integer multiples of a *scheduling quantum* Q (hence, a job may be delayed by up to Q time units before being scheduled).

Both approaches are illustrated in Fig. 1. Inset (a) shows an ideal, event-driven schedule of three jobs on two processors. The schedule is "ideal" in the sense that releases and completions are processed immediately (and require no processing time), and preemptions are enacted in zero time across processors. For example, at time 6.5, T_1^x is released on P_1 and immediately scheduled on P_2 without incurring any delay. While unattainable in practice, most G-EDF analysis assumes ideal, event-driven scheduling.

The same scenario in the ideal, quantum-driven case for $Q = 1$ is shown in Fig. 1(b). Again, the system does not incur overhead, but jobs are delayed when awaiting the next quantum boundary. For example, T_3^z is released at $r_3^z = 1.5$, but not scheduled until time 2 when the next quantum starts. Similarly, the preemption on P_2 due to T_1^x's arrival at $r_1^x = 6.5$ does not take place until the start of the next quantum at time 7. Note that some quanta may only be partially used if jobs complete during a quantum (*e.g.*, P_1 is partially idle for this reason in quantum [8,9)). Analysis assuming ideal, event-driven G-EDF scheduling can be applied to ideal, quantum-driven scheduling by shortening periods by one quantum (to account for delays upon release) and by rounding execution times up to a quantum multiple (to account for partially-idle quanta) [10].

Historically, OSs have employed quantum-driven (or hybrid) designs to facilitate time keeping and reduce overhead [16]; the first version of LITMUS$^{\mathrm{RT}}$ also followed this approach [9]. The current version supports both event-driven and quantum-driven scheduling.

LITMUS$^{\mathrm{RT}}$. As mentioned in Sec. 1, LITMUS$^{\mathrm{RT}}$ is a real-time extension of the Linux kernel.[5] The stock Linux scheduler is organized as a static hierarchy of *scheduling classes*: when the scheduler is invoked, each scheduling class is queried in top-down order until a process to service next is found. LITMUS$^{\mathrm{RT}}$ installs its scheduling class at the top of the hierarchy and hence overrides the stock Linux scheduler whenever real-time work is pending.

The LITMUS$^{\mathrm{RT}}$ scheduling class does not implement any particular scheduling policy; instead it allows scheduling policy plugins to be activated at runtime. All scheduling decisions are delegated by invoking plugin-provided *event*

[2]While LITMUS$^{\mathrm{RT}}$ supports real-time synchronization among tasks, this study is focussed on synchronization requirements *within* the kernel.

[3]We expect the reader to be familiar with the sporadic task model; see [10, 16] for an overview and relevant citations.

[4]We assume that priorities are unique, *i.e.*, that deadline ties are broken arbitrarily but consistently. LITMUS$^{\mathrm{RT}}$ tie-breaks by lower task index.

[5]The current base version is Linux 2.6.24. We plan to rebase LITMUS$^{\mathrm{RT}}$ to the latest kernel version in the near future.

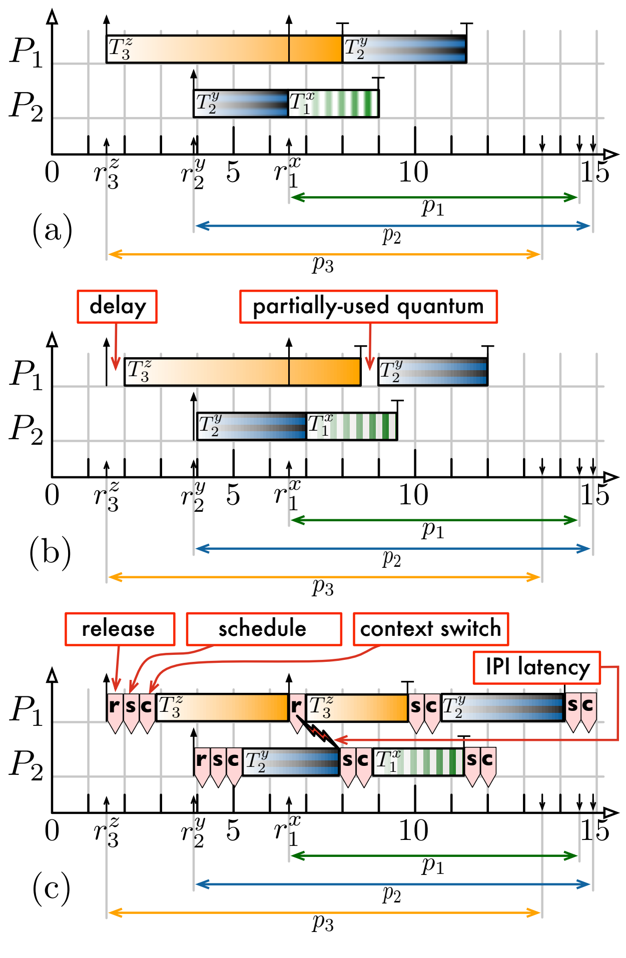

Figure 1: Example G-EDF schedules of three jobs (T_1^x, T_2^y, T_3^z), where $(e_i, p_i)_i = (2.5, 8)_1, (6, 11)_2, (6.5, 12)_3$, on two processors ($P_1$, P_2) illustrating delays introduced by quantum-driven scheduling and system overheads. Large up-arrows denote interrupts, small up-arrows denote job releases, down-arrows denote job deadlines, T-shaped arrows denote job completions, and wedged boxes denote overheads (which are magnified for clarity). Job releases occur at $r_3^z = 1.5$, $r_2^y = 3.9$, and $r_1^x = 6.5$. **(a)** Event-driven schedule and **(b)** quantum-driven schedule without overheads. **(c)** Event-driven schedule with overheads.

handlers, prominently among them the tick and schedule handlers. The tick handler is invoked, on each processor, each time a periodic timer interrupt occurs; this allows quantum-driven polices to be implemented. A typical tick period (*i.e.*, quantum length) is one millisecond. The schedule handler is invoked when traversing the scheduler class hierarchy to select the next process. Note that a quantum-driven plugin will make use of both the tick and schedule handlers—the tick handler assigns processes to processors and determines if preemptions are required, and the schedule handler, on each processor, enacts the desired changes. This split is required because the schedule handler, for technical reasons, *must* execute on the processor on which the preemption is to occur. In LITMUS$^{\text{RT}}$'s event-driven plugins, the tick handler is usually only used for bookkeeping and overrun detection.

As a boot-time option, LITMUS$^{\text{RT}}$ supports both *aligned* and *staggered quanta*. With aligned quanta, the per-processor tick interrupts are programmed to occur at the same time on all processors, whereas with staggered quanta, tick interrupts are spread out evenly across a full quantum. Staggering quanta may reduce bus and lock contention, but also delays job completions by up to $Q \cdot \frac{(m-1)}{m}$ time units. This can be accounted for by shortening periods.

Overheads. In LITMUS$^{\text{RT}}$, processes are delayed by six major sources of overhead, four of which are illustrated in Fig. 1(c). *Release overhead* is incurred while handling an interrupt that releases a real-time job and involves making the corresponding process available for execution. If a preemption is required, then *scheduling overhead* is incurred while selecting the next process to execute and re-queueing the previously-scheduled process (which may be a background or the idle process). *Context-switching overhead* is incurred while switching the execution stack and processor registers. All three overhead sources occur in sequence in Fig. 1(c) on processor P_1 at time 1.5 when T_3^z is released, and again on P_2 at time 3.9 when T_2^y is released. A different scenario occurs at time 6.5 when T_1^x is released on P_1: release overhead is incurred on P_1 (where the interrupt occurred), but scheduling and context-switching overhead are incurred on P_2 where T_1^x preempts T_2^y (the lowest-priority scheduled job). To initiate the required preemption, P_1 sends an *inter-processor interrupt* (IPI) to P_2. Since IPIs are not delivered instantly, T_1^x incurs additional *IPI latency*.

For the sake of clarity, Fig. 1(c) omits the two additional sources of overhead. At the beginning of each quantum, *tick overhead* is incurred *on each processor* when the periodic timer interrupt is handled. Note that tick overhead also occurs, but to a lesser extent, under purely event-driven plugins as the periodic tick can currently not be disabled in Linux (while a process is executing). *Preemption* and *migration overhead* account for any costs due to a loss of cache affinity. Preemption (resp., migration) overhead is incurred when a preempted job later resumes execution on the same (resp., a different) processor.

Analysis that assumes ideal (*i.e.*, overhead-free), event-driven scheduling can be applied to real, overhead-impacted systems by inflating per-task worst-case execution costs. Accounting for overheads that only occur exactly before or after a job is scheduled is trivial as they are, from an analysis point of view, equivalent to extended execution: each job causes scheduling and context-switching overhead exactly twice [16], and causes preemption/migration overhead at most once.[6] Similarly, a job's completion time may be delayed by IPI latency and its *own* release overhead.

However, accounting for release overhead due to *other* jobs and tick overhead is more problematic as their occurrence is interrupt-based and hence not subject to G-EDF scheduling [8]. For example, in Fig. 1(c), T_3^z is delayed by release overhead at time 6.5 due to T_1^x's release even though T_3^z has higher priority than T_1^x. As the techniques for multiprocessor interrupt accounting are somewhat involved, a detailed discussion is unfortunately beyond the scope of this paper; the interested reader is referred to [8].

Given the preceding discussion, we can now refine the focus of this paper: we seek to understand how differences in the implementation of global policy plugins affect runtime overheads, and hence, after accounting for overheads, real-time performance (in terms of "schedulability"—see Sec. 4).

3 Plugin Implementation

Conceptually, a global scheduling policy implementation consists of three main components: a *release queue* holds not yet released jobs; a one-to-one *processor mapping* associates each of the currently-scheduled jobs with a processor; and pending jobs that are not currently scheduled are kept, sorted by descending priority, in a shared *ready queue*. Since these components are shared by all processors, the way in which *synchronization* is provided strongly impacts overheads. Next, we briefly describe some implementation and synchronization choices for these components.

Release queue. A release queue is required for two reasons: time-driven tasks (*e.g.*, video display) require job releases to occur at particular times, and interrupt-driven tasks (*e.g.*, sensor data acquisition) may have jobs triggered "too early" after the last job release, *i.e.*, the minimum job separation may have to be enforced by the OS. In both cases, a job must be made available for execution at a future point in time. In a quantum-driven implementation, the release queue can be implemented as a priority queue or timer wheel [18] that is polled by the tick handler to transfer all jobs with release times in the preceding quantum to the ready queue. Alternatively, hardware timers can be programmed to trigger future job releases with interrupts if sufficiently high-resolution hardware timers are available.

As our test machines have such timers, LITMUS$^{\text{RT}}$ follows the latter approach. However, instead of using a timer for every job, our implementation uses a timer per release time to avoid unnecessary overhead, *i.e.*, if multiple job releases coincide (*e.g.*, on a hyperperiod boundary), then only one timer interrupt is required. Efficient timer sharing is accomplished by looking up future release times in a hash table. As a side effect, timer sharing enables the use of mergeable queues (see below).

Processor mapping. Since scheduling and switching between processes takes time, the notion of when a job is "scheduled" is not clear-cut. For example, consider a scenario in which a job T_i^j is one of the m highest-priority jobs when released, but a higher-priority job T_k^l arrives before the context switch to T_i^j's implementing process is complete. Now suppose that a third job T_x^y with priority less than T_k^l's but higher than T_i^j's arrives concurrently on another processor. Which job is "scheduled?" Is a preemption required? Should an IPI be sent?

In our experience, relying on the OS's notion of whether a particular *process* is scheduled (*i.e.*, is its stack in use?) for assigning *jobs* to processors is both difficult and error-prone, and tends to lead to priority inversions due to unanticipated corner cases and race conditions.[7] Instead, in LITMUS$^{\text{RT}}$, we use the processor mapping to split *job scheduling*, which is at the level of the sporadic model and is only concerned with assigning abstract jobs to abstract processors, from *process scheduling*, which is concerned with address spaces, stacks, register sets, and other hardware peculiarities.

This simplifies the real-time scheduling logic since any processor's job assignment can be updated on any processor by any event handler (in contrast to performing context switches, which can be only done in one specific code path on the target processor). Process scheduling is then reduced to tracking which process *should* be executing based on the current job-to-processor mapping; any delay in tracking is captured by the various overhead terms.

Since the most common operation involving the processor mapping is to check whether a preemption is required, which requires identifying the lowest-priority scheduled job, the processor mapping is realized as a min-heap with processors ordered by the priority of their assigned jobs (if any—idle processors have the lowest priority).

Ready queue. In earlier versions of LITMUS$^{\text{RT}}$, ready queues were realized as ordered linked lists. Not surprisingly, this simple approach did not scale well, and binomial heaps were employed instead [7]. Binomial heaps were chosen since they, together with timer sharing, support releasing jobs with coinciding release times in $O(\log n)$ time by means of a queue-merge operation.

However, since a binomial heap is a sequential data structure, it suffers from two inherent weaknesses when used as a shared queue on a multiprocessor: first, all ac-

[6]Since jobs starting to execute do not have cache affinity, only the preempted job (if any) must be considered.

[7]This is especially true when jobs may self-suspend for short durations (due to blocking on semaphores or page faults, library loading, *etc.*). However, handling self-suspensions is beyond the scope of this paper.

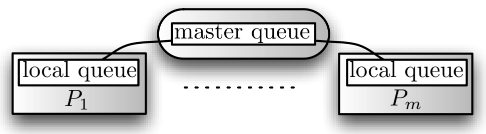

Figure 2: Illustration of hierarchical queues.

cesses must be synchronized through the use of a lock, which is likely to incur high contention; and, second, accesses are likely to cause significant cache-coherency traffic due to cache lines "bouncing" among processors.

To investigate the impact (if any) of these shortcomings, we consider two alternate priority-queue designs in this paper. With regard to lock contention, we implemented Hunt *et al.*'s concurrent heap [14, 15], which relies on fine-grained locking to increase parallelism. To address "cache line bouncing," we implemented a simple hierarchical priority queue that relies on (mostly) local per-processor queues to increase cache locality. The local queues contain the bulk of the jobs, and the global queue only consists of the highest-priority job on each processor (illustrated in Fig. 2). Under this scheme, insertion and merge operations are always performed on the cache-local queue, and an update of the global queue is only required if the highest-priority job changes in the local queue. However, removal of the highest-priority job from the global queue may require non-cache-local updates. Obviously, this algorithm is closely tied to the assumption that insertion and merge operations are evenly distributed across processors. (We used binomial heaps for both local and global queues.)

Another approach to avoiding lock contention is to employ *non-blocking* data structures, which allow accesses to occur concurrently (see [14] for an overview and relevant citations). We did not implement non-blocking priority queues for this study since, to the best of our knowledge, all such published algorithms only support bounded priority ranges (G-EDF requires unbounded priorities) or rely on problematic techniques such as multi-word compare-and-swap instructions (not supported by our hardware), frequent copying (excessive overheads), and probabilistic algorithms (ill-suited to real-time computing).

To summarize, the three ready-queue choices considered in this paper are:

R1 a sequential binomial heap (coarse-grained locking);

R2 Hunt *et al.*'s fine-grained heap [15];

R3 a simple, hierarchical "queue of queues" scheme (two-level locking with cache-local insertions).

Synchronization. In the latest publicly-available LITMUS$^{\mathrm{RT}}$ version (2008.2), implementation R1 is used. In this version, the processor mapping is protected by a lock that serves a dual purpose: it protects the (sequential) heap against concurrent updates, and it serves as the *linearization point* (see [14]) for the scheduler. Hence, it is acquired when the processor mapping itself is modified and whenever process state is observed or changed (*e.g.*, on process self-suspension, on job completions, and when comparing job priorities). This strategy also indirectly supports R1: all ready-queue accesses occur together with preemption checks, which require the processor mapping lock to be held.

However, to support R2 and R3, significant changes were required, as the processor mapping lock must be released before performing queue operations under these schemes. In making these changes, various complex race conditions had to be addressed. For example, both the ready queue and processor mapping must be locked to check whether a preemption is required. If a preemption is indeed required, then the processor mapping lock must be dropped before dequeuing the preempting job under R2 and R3. However, this allows the set of scheduled jobs to change while the preempting job is dequeued, which might lead to conflicts with other updates. To avoid this, the processor to be updated is temporarily removed from the processor mapping. Priorities are checked again upon re-insertion, otherwise, preemptions could be missed.

Ideally, the processor mapping lock should be held as little as possible to reduce contention. Unfortunately, the need to serialize (complex) process state updates makes a switch to fine-grained locking or non-blocking solutions non-trivial.

The release queue is also protected by a lock. However, contention for it is infrequent (it is only held briefly during job releases, after job completions, and possibly when a process resumes).

Interrupt handling. Two interrupt handling choices are considered in this paper:

I1 *global interrupt handling*, wherein each processor both handles interrupts and schedules jobs, and

I2 *dedicated interrupt handling*, wherein a single processor is reserved for interrupt processing [17].

By default, interrupts may occur on all processors in Linux. In fact, an even distribution of interrupts can help to improve throughput in non-real-time workloads. However, this implies that all real-time tasks are subject to release overhead (and delays by other interrupt sources, such as I/O devices). Since interrupt accounting can be severely pessimistic [8], it may be desirable to shield tasks from interrupts by dedicating a processor that does *not* serve tasks to handling job releases (and other interrupts).

While simple in concept, this approach can be difficult to implement if some hardware is only accessible from a particular processor. For example, in our test platform (see

Plugin	Ready Queue	Scheduling	Interrupts	Capacity
CQm	coarse-grained	quantum-driven	m	m
CEm	coarse-grained	event-driven	m	m
FEm	fine-grained [15]	event-driven	m	m
HEm	hierarchical	event-driven	m	m
CQ1	coarse-grained	quantum-driven	1	$m-1$
CE1	coarse-grained	event-driven	1	$m-1$
FE1	fine-grained [15]	event-driven	1	$m-1$

Table 1: Policy plugins evaluated in this paper. The letters of each plugin name refer to columns 2–4 in this table. An "Interrupts" value of 1 denotes the use of a dedicated interrupt processor. "Capacity" is the number of processors that schedule real-time tasks.

Sec. 4), each processor has a private, integrated hardware timer that is not accessible to other processors. This is resolved by sending IPIs to the dedicated processor to initiate the programming of hardware timers.

Implemented approaches. We implemented seven of the twelve possible combinations of choices S1, S2, R1–R3, I1, and I2 in LITMUS$^{\mathrm{RT}}$, as listed in Table 1. Under event-driven scheduling, we considered all three queue variants with global interrupt handling (CEm, FEm, and HEm), and coarse- and fine-grained queues with dedicated interrupt handling (CE1 and FE1). Hierarchical queues were not considered in combination with dedicated interrupt handling since they rely on insertions occurring on all processors. Only coarse-grained queues were considered under quantum-driven scheduling (CQm and CQ1) because quantum boundaries act as implicit barriers and ready queues are only accessed at quantum boundaries. Hence, parallel access would only yield minimal gains (if any).

4 Experiments

To evaluate the implemented approaches, we conducted extensive schedulability experiments under consideration of runtime overheads as incurred in LITMUS$^{\mathrm{RT}}$ on a Sun UltraSPARC T1 "Niagara" multicore platform. The Niagara is a 64-bit machine containing eight cores on one chip running at 1.2 GHz. Each core supports four hardware threads,[8] for a total of 32 logical processors. On-chip caches include a 16K (resp., 8K) four-way set associative L1 instruction (resp., data) cache per core, and a shared, unified 3 MB 12-way set associative L2 cache. Our test system is configured with 16 GB of off-chip main memory. In contrast to Sun's proposed Niagara-successor "Rock," our first-generation Niagara does not employ advanced cache-prefetching technology.

While the Niagara is clearly not an embedded systems processor, it is nonetheless an attractive platform for forward-looking real-time systems research. Its power-friendly combination of many simple and slow cores, predictable hardware multi-threading, and a small shared cache is likely indicative of future processor designs targeting computationally-demanding embedded systems.[9] Thus, we believe any limitations exposed on the enterprise-class Niagara today to be of value as guidance to future embedded system designs. However, note that specific results, as with all implementation-based studies, only apply directly to the tested configuration.

Next, we briefly discuss how we measured overheads, and then present our study in detail.

4.1 Runtime Overheads

We used the same methodology to determine overheads as in earlier LITMUS$^{\mathrm{RT}}$-based studies (*e.g.*, [7]). Runtime overheads were obtained by enabling aligned quanta and measuring the system's behavior for periodic task sets consisting of between 50 and 450 tasks in steps of 50. For each plugin and task-set size, we measured ten task sets generated randomly (with uniform light utilizations and moderate periods; see Sec. 4.2 below), for a total of 90 task sets per scheduling algorithm. Each task set was traced for 30 seconds. We repeated the same experiments with staggered quanta. In total, more than 100 GB of trace data and 640 million individual overhead measurements were obtained during more than ten hours of tracing. After removing outliers, we computed for each plugin average and worst-case overheads as a function of task set size (for both aligned and staggered quanta), which resulted in 20 graphs. Due to space constraints, we only discuss the three representative graphs shown in Fig. 3 here; all graphs and per-plugin overheads are provided in an extended version of this paper [6].

Fig. 3(a) shows average release overhead under staggered quanta. One can clearly distinguish between the quantum-driven plugins CQm and CQ1, which incur only very little, constant overhead, and the other event-driven plugins, which suffer the effects of increasing contention and ready-queue lengths. Releases encounter less contention in quantum-driven plugins because released jobs are placed in a temporary queue that is merged at the next quantum boundary by the tick handler [10]. Note that the FEm and FE1 plugins incur significantly higher release overhead than the CEm and CE1 plugins. This is likely due to Hunt *et al.*'s fine-grained heap not supporting efficient queue-merge operations (*i.e.*, merges require $O(n \log n)$ time), increased cache protocol traffic (as queue elements are being accessed concurrently), and frequent locking operations. Similar reasoning applies to the HEm plugin.

Fig. 3(b) shows average scheduling overhead under stag-

[8]The Niagara's hardware threads are real-time-friendly because each core distributes cycles in a round-robin manner—in the worst case, a hardware thread can utilize every fourth cycle.

[9]Similarly, 32-bit processors with L2 caches and speeds in excess of 100 MHz, once firmly associated with enterprise-class servers, are now routinely deployed in embedded systems.

gered quanta. Again, quantum-driven plugins incur only little overhead because staggering helps to avoid contention, and because jobs are assigned to processors in the tick, and not the schedule, handler. Staggering does not affect when event-driven plugins schedule, so contention remains high. Note that the FEm, FE1, and HEm plugins incur only a fraction of the CEm plugin's average scheduling overhead because the processor mapping lock is relinquished during queue operations. The trends are reversed in Fig. 3(c), which depicts average tick overhead under staggered quanta: event-driven plugins only perform bookkeeping activities in the tick handler and hence incur only little overhead, whereas the CQm and CQ1 curves reveal that job scheduling costs increase with the number of tasks.

We used monotonic piece-wise linear interpolation to derive upper-bounds for each plugin and each overhead as a function of task set size. These upper bounds were used in the schedulability experiments described next.

4.2 Experimental Setup

To assess schedulability trends, we generated random task sets (similarly to [7, 9]) using three period and six utilization distributions similar to those proposed by Baker [2], for a total of 18 scenarios. Task utilizations were distributed differently for each experiment using three uniform and three bimodal distributions. The ranges for the uniform distributions were [0.001, 0.1] (*light*), [0.1, 0.4] (*medium*), and [0.5, 0.9] (*heavy*). In the three bimodal distributions, utilizations were distributed uniformly over either [0.001, 0.5) or [0.5, 0.9] with respective probabilities of 8/9 and 1/9 (*light*), 6/9 and 3/9 (*medium*), and 4/9 and 5/9 (*heavy*). Similarly, we considered three uniform task period distributions with ranges $[3ms, 33ms]$ (*short*), $[10ms, 100ms]$ (*moderate*), and $[50ms, 250ms]$ (*long*). Note that all periods were chosen to be integral.

Task execution costs excluding overheads were calculated from periods and utilizations (and may be non-integral). Each task set was created by generating tasks until a specified cap on total utilization (that varied between 1 and 32) was reached and by then discarding the last-added task, thereby allowing some slack for overheads. Sampling points were chosen such that sampling density is high in areas where curves change rapidly. For each scenario and each sampling point, we generated 1,000 task sets, for a total of over 5.5 million task sets.

Schedulability tests. After a task system was generated, its schedulability under each of the plugins listed in Table 1 was tested as follows.

Prior to testing schedulability, we adjusted task parameters to account for overheads and quantum-driven scheduling as described in Sec. 2. Given Linux's roots as a general-purpose OS and our use of measured overheads (which are

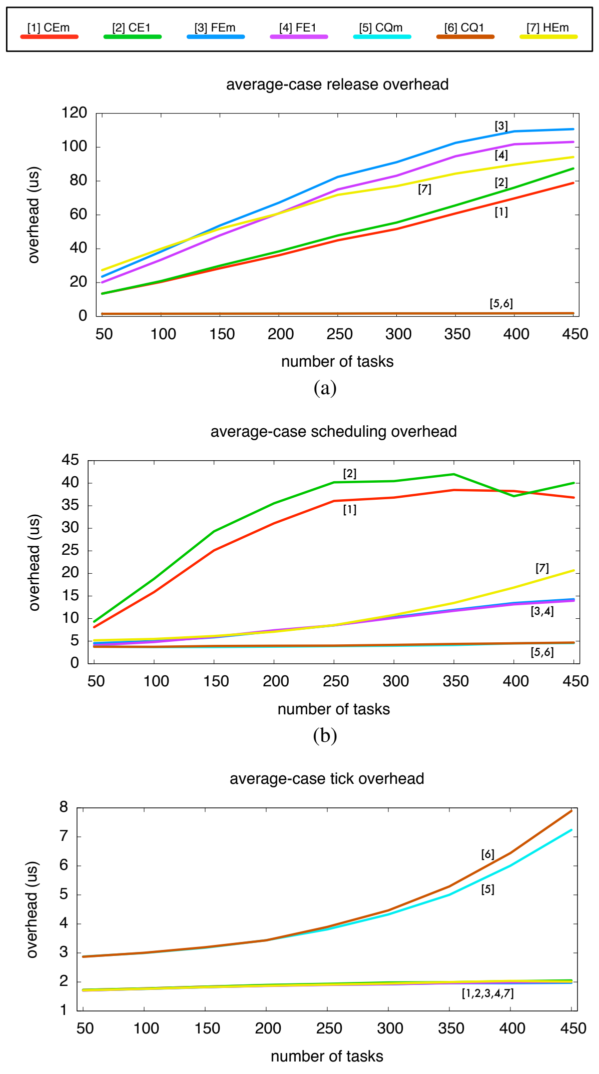

Figure 3: Sample average-case overhead measurements. The graphs show average-case measured per-event execution time (in microseconds) as a function of task set size (under staggered quanta). **(a)** Release overhead. **(b)** Scheduling overhead. **(c)** Tick overhead. Note the different scale of the Y-axis in each inset. The complete worst-case and average-case overhead measurements are reported in [6].

unlikely to capture true worst-case behavior), we interpret task execution costs in a way that is reasonable for a Linux-based system. Our main concern here (since this is not a paper on timing-analysis tools for determining execution costs) is to capture major differences in plugin implementations. Thus, in reality, we interpret "hard real-time" to mean deadlines should *almost never* be missed and "soft real-time" to mean that deadline tardiness *on average* remains bounded. Consequently, we assumed worst-case overheads and aligned quanta when testing HRT schedulability and average-case overheads and staggered quanta when testing SRT schedulability. Additionally, we also considered

HRT schedulability under the CQm and CQ1 plugins assuming worst-case overheads under staggered quanta; those two cases are denoted S-CQm and S-CQ1 respectively.[10] (Note that periods are not modified in the SRT case under quantum-driven scheduling as quantum-based delays cause only constant tardiness.)

We used all major published sufficient (but not necessary) HRT schedulability tests for G-EDF [1, 3, 4, 5, 12] and deemed a task set schedulable if it passed at least one of these five tests. A notable exception is the light uniform utilization distribution: we had to disable Baruah's test [3] for these scenarios since it failed to terminate in reasonable time due to the large number of tasks involved and the test's pseudo-polynomial nature.[11]

For SRT schedulability, since G-EDF can guarantee bounded deadline tardiness if the system is not overloaded [11], only a check that total utilization (after inflation for overheads) is at most m is required.

4.3 Results

HRT schedulability results for the moderate period distribution are shown in Fig. 4. The first column of the figure (insets (a,c,e)) gives results for the three uniform distributions (light, medium, heavy) and the second column (insets (b,d,f)) gives results for the three bimodal distributions. The plots indicate the fraction of the generated task sets each plugin successfully scheduled, as a function of total utilization. Schedulability results for the SRT case are shown in Fig. 5, which is organized similarly to Fig. 4. Due to space constraints, other graphs (over 300) are not shown here but can be found in the extended version of the paper. [6]

The results clearly show that differences in plugin implementation and hence overheads can have a very significant impact on real-time performance. For example, in Fig. 5(a), the best-performing plugin (FE1) supports task systems of total utilization up to 26, whereas the worst-performing plugin (HEm) fails even for task systems with total utilization less than 5! Note that it is nearly impossible to predict such outcomes purely based on theoretical considerations. In fact, our initial assumption was that a hierarchical queue design should outperform any dedicated-interrupt scheme—surprisingly, the HEm plugin was consistently among the poorest-performing plugins. We will discuss this and other notable trends next.

Dedicated vs. global interrupt handling. The most remarkable trend is the superiority of dedicated interrupt handling. In virtually all tested scenarios, global interrupt handling is inferior in spite of its nominally larger system capacity, *i.e.*, CE1 outperforms CEm, FE1 outperforms FEm, and CQ1 outperforms CQm, in both the HRT and SRT cases, and usually by a significant margin (*e.g.*, Figs. 4(a,c,e) and 5(a–f)). Differences are less pronounced if periods are long (since overheads are proportionally small), or if performance is indistinguishably bad (*e.g.*, CQm/CQ1 for short periods).

Long periods with heavy, uniform utilizations is the only scenario in which global interrupt handling is consistently preferable—here, tasks are so few in number, periods so long, and overheads so low that release overhead becomes insignificant and system capacity is the deciding factor.

Staggered vs. aligned quanta. Another clear trend in the HRT results is that staggered quanta are generally preferable to aligned quanta, *i.e.*, S-CQm outperforms CQm and S-CQ1 outperforms CQ1 in most cases, as can be seen in Figs. 4(a,c,e). (This matches similar trends observed in the preceding scalability study [7].) Again, differences are less pronounced for high task count / short period scenarios (both plugins perform equally badly), and for heavy bimodal utilization / long periods (both plugins nearly reach the ideal G-EDF limit). As S-CQ1 is generally preferable to S-CQm (see above), the S-CQ1 plugin is the best-performing quantum-driven plugin in this study.

Quantum- vs. event-driven scheduling. In the HRT case, event-driven scheduling is mostly preferable. CQm outperforms CEm only in the cases of uniform light utilizations and moderate or long periods (Fig. 4(a)), and medium utilizations with moderate periods (Fig. 4(c)). Similarly, CQ1 is never preferable to CE1, though differences are small for some scenarios involving long periods. Further, S-CQm and S-CQ1 are inferior to CEm except for scenarios with many light tasks and long periods. Surprisingly, the trend is reversed in the SRT case. In Fig. 5, CQm is clearly preferable to CEm in every inset but (b), and similar findings apply to the case of short periods, too (but CEm remains superior for long periods). However, CE1 remains preferable to CQ1 for all period and utilization distributions, even in the SRT case. Hence, event-driven scheduling was found in our experiments to be the better choice for both HRT and SRT if (and only if) used in conjunction with a dedicated interrupt processor.

Fine-grained vs. coarse-grained vs. hierarchical queues. As mentioned above, hierarchical queues exhibited surprisingly poor performance in our experiments. In fact, in the SRT case, HEm is the worst-performing plugin in all tested scenarios, as can be readily seen in Fig. 5. It performs better in the HRT case (*e.g.*, Fig. 4(f)), but it is never preferable to the CEm plugin.

The FEm plugin is never preferable to the CEm plugin

[10] Only quantum-driven plugins were considered under staggering in the HRT case since neither overheads nor analysis are affected significantly by staggering under event-driven plugins.

[11] Regular compute jobs on UNC's research cluster are limited to 48 hours runtime. With, on average, over 150 sampling points per graph, 1,000 task sets per point, and 10 curves per HRT graph, *all five* HRT G-EDF schedulability tests must complete in ≈ 0.11 seconds per task set.

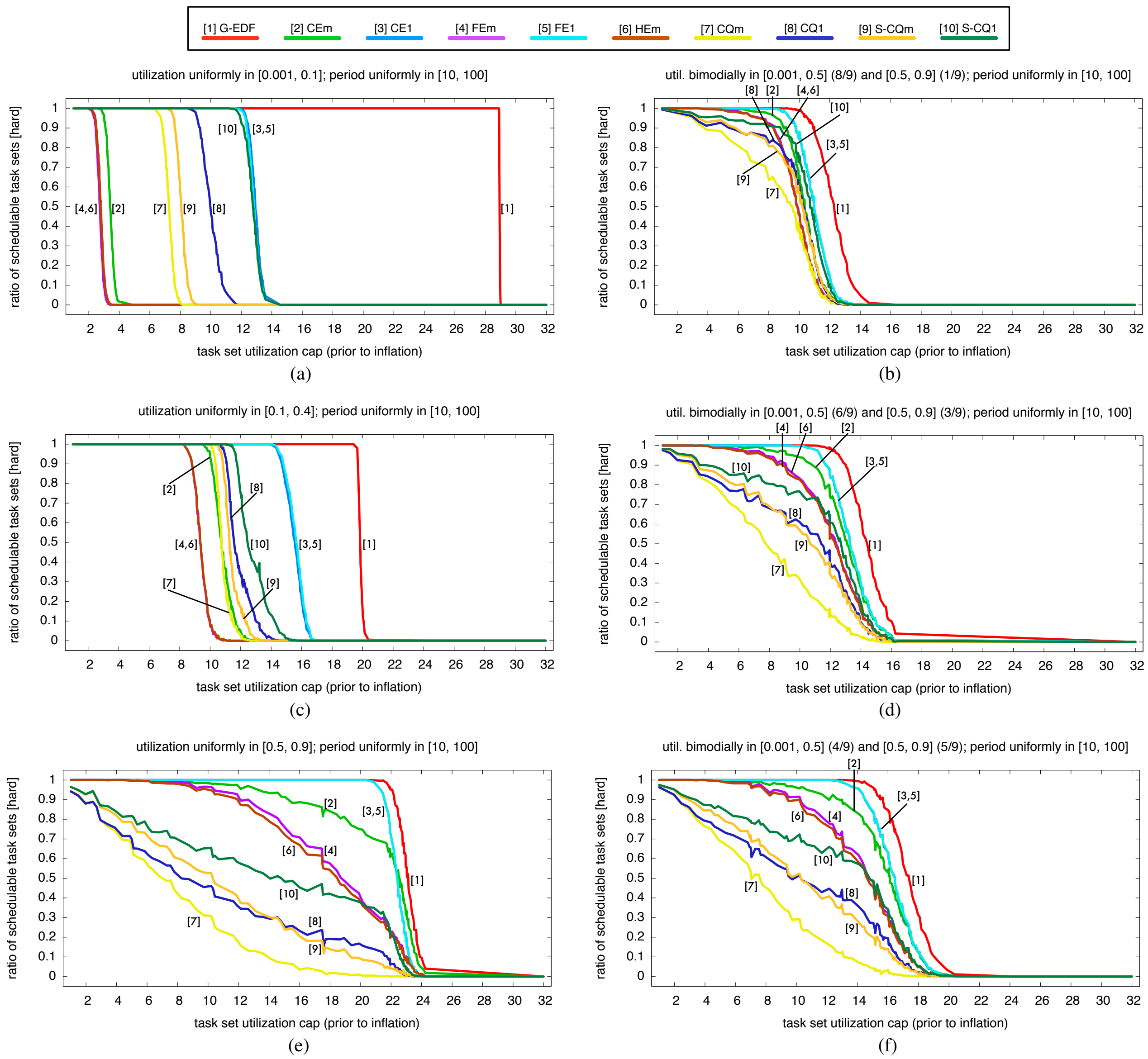

Figure 4: HRT schedulability (the fraction of generated task systems deemed HRT schedulable) for moderate periods as a function of task system utilization cap for various utilization distributions. **(a)** Uniform light. **(c)** Uniform medium. **(e)** Uniform heavy. **(b)** Bimodal light. **(d)** Bimodal medium. **(f)** Bimodal heavy. In each graph, the curve labeled "[1] G-EDF" indicates schedulability under an ideal event-driven system (zero overheads), the curves numbered 2–8 correspond to the plugins listed in Table 1, and the curves 9–10 correspond to curves 7–8 but assume staggered quanta. Recall from Sec. 4.2 that a task set is HRT schedulable if it can be shown to have *zero* tardiness under assumption of *worst-case* overheads.

in any of the tested scenarios. In the SRT case, the CEm plugin performs better by a significant margin in all scenarios, as is apparent in Fig. 5; in the HRT case, the difference is often less pronounced but still clear (*e.g.*, Fig. 4(f)). Interestingly, the FE1 plugin is very competitive—it never performs worse than the CE1 plugin, and in some scenarios marginally better. This highlights two points: reducing contention for the processor mapping lock is highly beneficial, but the lack of an efficient queue-merge operation penalizes FEm (see Fig. 3(a)).

Further observations. Overall, the FE1 and CE1 plugins performed best in our study. Their performance is promising in two regards. First, the disappointing G-EDF performance in [7] was shown here to be a correctable implementation artifact. Second, in many scenarios, FE1 and CE1 performance comes close to the limit imposed by current G-EDF schedulability analysis (*e.g.*, Fig. 4(b,d,e,f), and similarly Fig. 5(b–f)). This indicates that G-EDF in particular, and global scheduling in general, can be implemented efficiently on current multicore platforms.

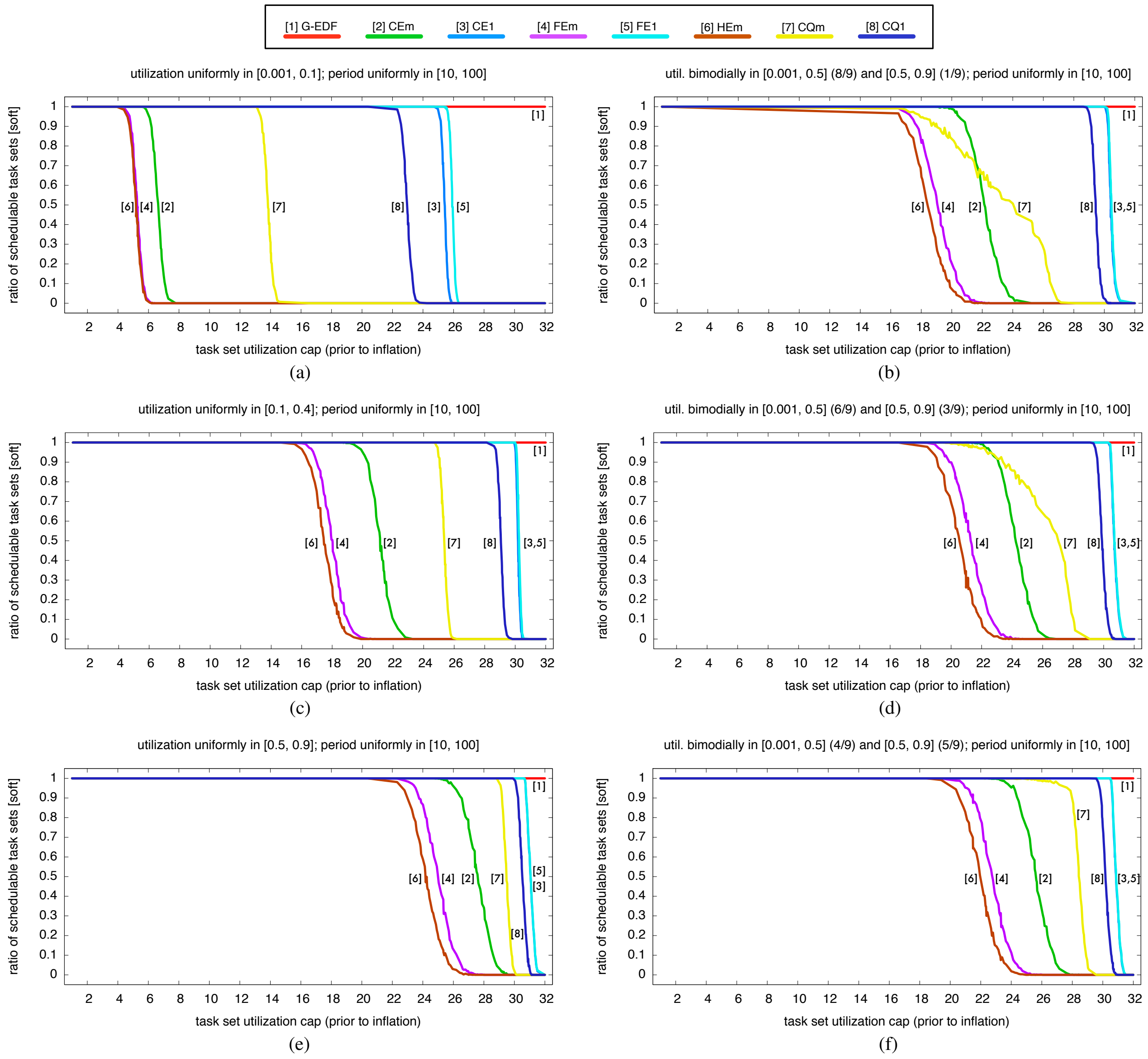

Figure 5: SRT schedulability (the fraction of generated task systems deemed SRT schedulable) for moderate periods as a function of task system utilization cap for various utilization distributions. **(a)** Uniform light. **(c)** Uniform medium. **(e)** Uniform heavy. **(b)** Bimodal light. **(d)** Bimodal medium. **(f)** Bimodal heavy. In each graph, the curve labeled "[1] G-EDF" indicates schedulability under an ideal event-driven system (zero overheads) and the curves numbered 2–8 correspond to the plugins listed in Table 1. Recall from Sec. 4.2 that a task set is SRT schedulable if it can be shown to have *bounded* tardiness under assumption of *average-case* overheads.

5 Conclusion

In this work, we explored several alternatives for implementing global real-time schedulers and conducted a large-scale implementation-based study. Our results indicate that implementation tradeoffs can impact schedulability as profoundly as scheduling-theoretic tradeoffs. On our test platform, the best performance was achieved in the majority of the tested scenarios by the combination of Hunt *et al.*'s fine-grained heap [15], event-driven scheduling, and dedicated interrupt handling (the FE1 plugin). In contrast, the simple hierarchical queue (the HEm plugin) performed significantly below expectation—this partially due to cache-affinity issues, which warrant further investigation generally. Additionally, we found that the scalability of global real-time scheduler implementations *is not the key limiting factor* in supporting sporadic real-time workloads on our—fairly large—multicore platform unless task counts are extremely high or most periods short.

Prior studies. Given this paper's conclusion that implementation choices can have a strong impact on schedulability, it is appropriate to re-visit the conclusions of the two preceding studies [7, 9]. Since the first study [9] was conducted on a radically different hardware platform, its results are not directly comparable to this paper. However, its main conclusion that "for each tested scheme, scenarios exist in which it is a viable choice" [9] remains valid even with the improvements presented in this paper—while significantly better, G-EDF's HRT performance is still limited by pessimistic schedulability tests and contention under high task counts. Further, note that dedicated interrupt handling is likely less competitive on platforms with fewer processors.

The second study [7] is Niagara-based and hence readily comparable; the interested reader is encouraged to compare Figs. 4 and 5 above with Figs. 2 and 3 in [7]. As expected, the observation that "for global approaches, scheduling overheads are greatly impacted by the manner in which run queues are implemented" [7], which motivated the present study, has been validated. The relative performance of CQm vs. S-CQm (best seen in [6]) also confirms that "quantum-staggering can be very effective in reducing [preemption and migration] costs" [7], but the benefit observed in this study is less pronounced than that in [7].

Further, as noted above, G-EDF's HRT performance is still limited, and hence the observation that "for HRT workloads on the Niagara, [staggered Pfair] and [partitioned EDF] are generally the most effective approaches" [7] remains unchanged. However, [7] also stated that, "for SRT workloads on the Niagara, there is no single best overall algorithm, although [staggered Pfair] and [clustered EDF] seem to be less susceptible to pathological scenarios than the other tested algorithms." This statement has to be amended to include G-EDF, as the FE1 plugin is consistently among the top-performing choices (with regard to those in [7]) in the SRT case.

Future work. Our implementation efforts and the results of our study reveal four major open problems. From an analysis point of view, the two most-pressing concerns are the need for improved G-EDF HRT schedulability tests (ideally, both in accuracy and runtime requirements) and improved interrupt accounting techniques. Implementation-wise, better parallel priority queues that can be implemented efficiently in a kernel environment clearly have great potential. Ideally, such queues should efficiently support queue-merge operations. Further, fine-grained locking (or even non-blocking synchronization) for the processor mapping could improve performance in the presence of many tasks significantly. Especially with regard to the latter, we would like to investigate whether the hardware transactional memory support in Sun's proposed "Rock" processor can improve global schedulers.

References

[1] T. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, 2003.

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

[3] S. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 119–128, 2007.

[4] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, 2005.

[5] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, 2009.

[6] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. Extended version of this paper. Available at http://www.cs.unc.edu/˜anderson/ papers.html.

[7] 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, 2008.

[8] B. Brandenburg, H. Leontyev, and J. Anderson. Accounting for interrupts in multiprocessor real-time systems. In *Proceedings of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 273–283, 2009.

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

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

[11] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

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

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

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[15] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.

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

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

[18] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: data structures for the efficient implementation of a timer facility. *SIGOPS Operating Systems Review*, 21(5):25–38, 1987.