Scaling Global Scheduling with Message Passing

Felipe Cerqueira Manohar Vanga Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Global real-time schedulers have earned the reputation of scaling poorly due to the high runtime overheads involved in global state management. In this paper, two mature implementations, one using fine-grained locking (SCHED_DEADLINE) and one using coarse-grained locking (LITMUS$^{\mathrm{RT}}$'s G-EDF plugin), are evaluated and it is shown that, regardless of locking granularity, indeed neither scales well w.r.t. worst-case overheads due to excessive lock contention. To demonstrate that this is not an inherent limitation of global scheduling, the design of G-EDF-MP is presented, a global scheduler that uses message passing to avoid lock contention and cache-line sharing. It is shown to offer up to a 23- to 36-fold reduction in worst-case scheduling overhead on a 64-core platform, which translates into much improved schedulability (in some cases, more than 120 additional tasks can be supported).**

I. Introduction

Global real-time schedulers dispatch tasks to processors dynamically at runtime based on the state of all processors and all tasks. Such a global view of the system enables a wealth of attractive algorithmic properties—most famously optimality (*e.g.*, [6, 24, 37])—that have spurred considerable interest in global scheduling in recent years (*e.g.*, [2, 5, 10, 26, 28, 31], see [23] for a recent overview). However, in practice, the Achilles' Heel of global scheduling is runtime overheads [7, 13, 16, 18]: maintaining the required consistent global state can be costly, and becomes increasingly problematic with rising core counts. One might thus reasonably wonder: is it at all possible to *scale* global scheduling beyond a handful of cores?

In this paper, we answer this question in the affirmative. We first study two mature, radically different implementations of global real-time scheduling—LITMUS$^{\mathrm{RT}}$'s *global earliest-deadline first* (G-EDF) scheduler [13, 17] and the SCHED_DEADLINE scheduler for stock Linux [27, 32, 33] (SD hereafter)—and show that they indeed do *not* scale. LITMUS$^{\mathrm{RT}}$'s implementation uses a single *coarse-grained* lock to protect all scheduler state; its lack of scalability is thus expected. However, to our surprise, we found that SD, which uses *fine-grained* per-processor locks, does not scale either: it achieves much lower average-case overheads than LITMUS$^{\mathrm{RT}}$, but only at the expense of crippling worst-case overheads.

Based on the observation that the main scalability bottleneck is not locking granularity *per se*, but rather the number of processors that may access the same lock(s), we then sidestep the global state management problem with a novel design based on *message passing*. In our new design, processors report state changes to a dedicated scheduling processor that is responsible for *all* scheduling decisions. As a result, the global state is not shared, scheduling decisions are implicitly serialized, and no scheduler lock is ever accessed by more than two processors, a potent combination that scales well: on a 64-core Intel Xeon platform, we observed a 23x (resp., 36x) reduction in worst-case scheduling overhead w.r.t. LITMUS$^{\mathrm{RT}}$ (resp., SD).

A. Motivation: The Case for Global Scheduling

The defining property of a global scheduler is that each task may execute on potentially any processor, depending on availability and each task's current priority. Implicit in this definition is the reason why it is difficult to implement global scheduling: to dynamically dispatch tasks according to processor availability, and in accordance with the scheduling policy (*e.g.*, in order of increasing deadlines under G-EDF), *a globally consistent snapshot* of the entire system state is needed. As globally shared (and frequently changing) state is well known to be a scalability bottleneck, it is not surprising that global schedulers are inherently more afflicted by runtime overheads than partitioned schedulers (in which tasks are statically assigned to processors and all dispatching decisions can be made locally).

Nonetheless, there are good reasons to not give up on global scheduling. As already mentioned, (practical) optimal multiprocessor real-time schedulers are necessarily global. Further, even if optimality of the scheduler is a secondary concern (or none at all, as it is admittedly often the case in practice), global scheduling offers advantages that make it preferable for certain applications. For example, in *adaptive systems* (*i.e.*, if task requirements change at runtime in reaction to environmental changes) and in *open systems* (*i.e.*, if tasks may exit or join the system dynamically), global scheduling is much easier to deal with precisely because it dispatches tasks dynamically and hence avoids the need for complex task mapping adjustments [12].

Other reasons speaking for global scheduling are that it is less susceptible to deadline misses due to intermittent overloads (as the load is spread across all processors) [23], that it offers lower average-case scheduling latencies (because no processor idles while any task is waiting) [23], that it is best suited for race-to-idle energy conservation strategies [31], and that it permits much simpler locking mechanisms (*e.g.*, classic priority inheritance works as expected under global scheduling, but is ineffective under partitioned scheduling, see *e.g.* [13]).

To summarize, while global scheduling is not necessarily the best approach for *all* real-time workloads, there exist many applications that can benefit significantly from *efficient* OS support for global real-time scheduling for reasons of flexibility, optimality, and simplicity. Case in point: while Linux, QNX, and many other commercial real-time OSs offer flexible scheduling APIs that allow specifying arbitrary processor affinities [29], they *default* to global scheduling.

B. Focus of This Paper

Motivated by the overhead challenges surrounding global scheduling, our objective is to understand how global scheduling policies may be efficiently supported on multicore platforms that are large by today's standards.

To this end, we study two mature and actively maintained open-source versions of G-EDF in Linux as reference implementations reflecting the current state of the art. We focus on the G-EDF policy in this paper for a number of reasons. For one, it has received considerable attention in prior work [2, 5, 10, 23, 28] and multiple independent implementations are readily available. Further, it is well-suited for the (predominantly) soft real-time workloads likely to be encountered on large multicore platforms because it is optimal (in the sense that it is not subject to utilization loss) if bounded deadline tardiness is permitted [24]. And finally, it is representative of the larger class of *job-level fixed-priority* schedulers, which includes both fixed-priority scheduling (the default in most commercial real-time OSs such as QNX and VxWorks) and other interesting (soft) real-time schedulers [26]. In short, if it is possible to scale G-EDF, then many other interesting policies can be implemented similarly.

Contributions. The primary contribution of this paper is to demonstrate that it is indeed possible to construct scalable global schedulers: global scheduling is not *inherently* limited by kernel-level lock and cache contention, though it is the case with many existing implementations. To support this claim:

1) We first review the challenges inherent in global scheduling and the designs of SD and LITMUS$^{\mathrm{RT}}$'s G-EDF plugin, and identify key performance and correctness limitations in Sec. II.
2) To overcome the scalability limitations of the current lock-based solutions, we introduce a novel scheduler design in Sec. III that is primarily based on message passing, and in which the remaining locks are accessed in pairwise fashion only.
3) We demonstrate that our design performs significantly better at scale than either of the two major current G-EDF implementations by means of average- and worst-case overhead measurements across six multicore platforms with 8, 16, 24, 32, and 64 cores in Sec. IV. For instance, in the 64-core setup, the worst-case scheduling overhead is reduced by a factor of 23 compared to LITMUS$^{\mathrm{RT}}$'s G-EDF plugin, and by a factor of 36 compared to SD.
4) Finally, in Sec. V, we report on large-scale schedulability experiments that show that the reduction in overheads translates into significantly improved schedulability: when considering soft real-time schedulability (*i.e.*, bounded tardiness) in the 64-core setup, our message-passing-based implementation can support more than 120 additional real-time tasks than the best prior implementation.

A secondary contribution of this paper is the first direct comparison of LITMUS$^{\mathrm{RT}}$ and SD on the same hardware platform and with identical workloads, which is long overdue given their similarity in aim and differences in design.

We review related work in Sec. VI, and begin with a discussion of the scalability challenges posed by global scheduling.

II. Scalability Challenges

Before discussing the scalability limitations of existing implementations, we need to clarify the relevant metrics. In this paper, we focus on the scalability of *scheduling overheads*: the costs of making a scheduling decision and dispatching a task (discussed in detail in Sec. IV), which should grow only slowly (or ideally not at all) with increasing core counts.

Another source of overheads that affects tasks under global scheduling are *cache-related preemption and migration delays* (CPMD), which tasks incur due to a loss of cache affinity when they resume execution after a preemption or migration. While CPMD is a non-negligible source of overhead in practice (and accounted for in the experiments in Sec. V), it is less relevant in the context of this paper because CPMD is primarily hardware-dependent (*i.e.*, not affected by the kernel's scheduler implementation). Further, CPMD is not specific to global scheduling, but also affects partitioned (and semi-partitioned) scheduling, whereas overhead scalability is a challenge specific to global scheduling (partitioning is trivially scalable w.r.t. overheads).

In the following, we let m denote the number of processors, and assume familiarity with the sporadic task model on behalf of the reader. In short, a *sporadic task* releases a sequence of *jobs*, where each job corresponds to an invocation of the task and has an associated release time and deadline. From the point of view of the Linux kernel, a sporadic task is simply a process that more or less regularly becomes ready for scheduling. We therefore use the terms 'task' and 'job' interchangeably when discussing implementation matters. With these clarifications in place, we next discuss why it is difficult for global schedulers to maintain low overheads with increasing processor counts.

A. Requirements and Challenges

Implementing a correct global scheduler requires the maintenance or discovery of a consistent global snapshot of the system that includes **(i)** the current task-to-processor mapping (*i.e.*, which tasks are scheduled) and **(ii)** the set of jobs that are eligible to run but not scheduled (to determine whether a preemption is required). Such a consistent view is necessary to avoid *priority inversions*, that is, to ensure that the *global scheduling invariant* is satisfied at all times: in a system with m processors and r ready tasks, the $\min(m, r)$ highest-priority tasks must be scheduled, where each task's current priority is determined by a policy such as EDF.

Given a consistent snapshot, making a correct scheduling decision (*i.e.*, one that maintains the global scheduling invariant) is trivial. However, since scheduling events—job arrivals, completions, suspensions, and resumptions—may occur concurrently on any processor, the state on which a scheduling decision is based may be invalidated before the scheduler has finished making its decision. There are two possible ways to deal with this issue: serialize all scheduling events (as done in LITMUS$^{\mathrm{RT}}$), thereby avoiding concurrent state changes, or discard the invalidated decision and retry immediately (Linux and SD use this technique). Unfortunately, both approaches are problematic from a scalability perspective. Whereas the former approach, if implemented with a global lock, introduces an obvious contention hotspot, the latter can be problematic because concurrent state changes may race repeatedly (*i.e.*, a processor may have to retry many times before arriving at a consistent decision). Worse, both contention and the likelihood of concurrent state changes increase with the number of cores.

In fact, this very issue—maintaining a task-to-processor assignment that respects the global scheduling invariant in the face of concurrent scheduling events—is the root of most complexity and scalability issues in global schedulers.

Another, more analytical problem is posed by *job release interrupts*. Jobs are typically released (*i.e.*, in Linux, processes become ready) in response to timer or device interrupts. Such interrupts are thus an inherent aspect of global real-time scheduling; however, they are not controlled by the scheduler. Rather, interrupts are dispatched in hardware and always take precedence over the OS's scheduling policy. This deviation from the scheduling policy causes considerable analytical complications. Since most published schedulability tests do not account for interrupts (*e.g.*, see [23]), their delaying effects must be accounted for separately prior to normal schedulability analysis [19]. However, because it is in general not possible to predict on which processor(s) a task will execute, it must be assumed that, in the worst case, a job is delayed by all interrupts that occur while it is pending [13, 19], which introduces considerable pessimism. Minimizing the impact of job releases on already scheduled jobs is thus an important goal in the implementation of global schedulers [16].

To motivate the scalability problem further, and to illustrate the impact of the just-described key issues, we next review the two considered reference implementations and demonstrate their lack of scalability on a 64-core Intel Xeon platform.

B. The Design and Scalability of G-EDF in LITMUS^{RT}

LITMUS$^{\mathrm{RT}}$ [1], a real-time extension of the Linux kernel originally developed at UNC Chapel Hill, has included a G-EDF plugin since its initial version [20]. The version considered herein is called GSN-EDF because it was designed to support suspension-based locking protocols and non-preemptable sections with $O(1)$ priority inversions [11, 13]. GSN-EDF has been the main G-EDF implementation of LITMUS$^{\mathrm{RT}}$ since 2007 and has remained structurally unchanged since then. It has also been subject of several prior studies [7, 13, 16, 18, 20], which we revisit in the context of this work in Sec. VI.

The design of GSN-EDF is straightforward and favors worst-case predictability and conceptual simplicity over average-case optimizations: a *single* global spin lock protects both the shared ready queue (a binomial heap) and the processor mapping that links tasks to processors, and also serializes all scheduling decisions. This simple, coarse-grained synchronization approach has the advantage that it is relatively easy to reason about correctness, but obviously comes at a scalability tradeoff.

GSN-EDF's lack of scalability is readily apparent in Fig. 1, which shows an excerpt of the evaluation discussed in detail in Sec. IV. In short, we measured the average and maximum runtime overheads of GSN-EDF when managing 8, 16, 24, 32, 48, and 64 cores, respectively, of a 2 GHz 64-core Intel Xeon machine. Fig. 1 depicts the maximum and average costs of two key operations, namely the scheduling overhead (*i.e.*, assigning jobs to processors) and the release overhead (*i.e.*, adding jobs to the ready queue and triggering preemptions).

Both the maximum scheduling and release overheads exhibit a strong dependence on m and quickly become excessive beyond 16-24 cores, exceeding 1,000 μs and 600 μs, respectively, at 48 cores. Worse, this trend also manifests in the average-case overheads, which both reach about 500 μs at $m = 64$. Given that tasks with deadlines in the range of a few milliseconds are not uncommon, scheduling overheads of more than *one millisecond* are not viable and clearly show GSN-EDF's lack of scalability.

Prior studies using GSN-EDF [2, 10, 23, 28] used platforms with 4, 24, and 32 processors, a range in which GSN-EDF still exhibits overheads that are moderate in the context of Fig. 1. Nonetheless, in conjunction with the analytical limitations of current interrupt accounting techniques, the high release overheads proved problematic [16]. For this reason, GSN-EDF supports *dedicated interrupt handling* [16, 38], where a dedicated processor handles all release interrupts, and real-time tasks are scheduled only on the other $m - 1$ processors to shield them from interrupts. Given release overheads in the range of hundreds of microseconds (as evident in Fig. 1), this is a worthwhile tradeoff since the capacity lost to accounting-related pessimism far exceeds the "loss" of one processor [16].

C. The SCHED_DEADLINE Patch for Linux

The SD patch [27, 32, 33] for Linux was developed at the Scuola Superiore Sant'Anna in Pisa and fundamentally differs from GSN-EDF in both its design and its goal. While LITMUS$^{\mathrm{RT}}$ is primarily a prototyping framework and abstraction layer for experimental real-time research, SD is a comparably mature implementation that directly integrates with the kernel and aims at eventual inclusion into mainline Linux. Its design hence closely follows Linux's *push/pull* architecture, which is based on distributed, per-processor ready queues, each guarded by a separate spin lock, which reduces both lock and cache contention by minimizing data sharing across processors. GSN-EDF and SD thus represent two extreme opposites: while in GSN-EDF global coordination is the default (and all scheduling decisions are serialized), in the push/pull design, most scheduling decisions are made locally (and concurrently).

In Linux, a global policy such as G-EDF must be implemented on top of the per-processor ready queues by explicitly load-balancing the highest-priority tasks whenever local scheduling events require a global adjustment (*e.g.* when new jobs are released or when jobs complete). This is accomplished with the eponymous *push* and *pull* operations, which enact *source*- and *target-initiated* migrations, respectively, to maintain the global scheduling invariant.[1] Conceptually, pull operations scan all other ready queues to "steal" backlogged higher-priority tasks *before* a local scheduling decision is made, and push operations move local backlogged tasks to other processors where they can be scheduled immediately *after* a local scheduling decision was made. In the actual implementation, these operations are not invoked as part of every scheduler invocation; rather, processors push whenever a new job is released or a local job is preempted, and pull when a local job completes or suspends.

[1]While the Linux real-time scheduler defaults to global scheduling, it supports specifying *arbitrary processor affinities* to restrict a task's execution to a subset of processors. This allows emulating partitioned, global, and hybrid schedulers [29]. We focus on global affinities herein, as those are most difficult to support—adding affinity restrictions makes the (runtime) problem easier.

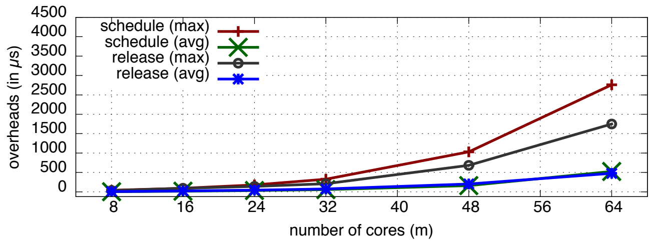


Fig. 1: Scheduling and release overheads under GSN-EDF. The average scheduling and release overheads coincide at this resolution.

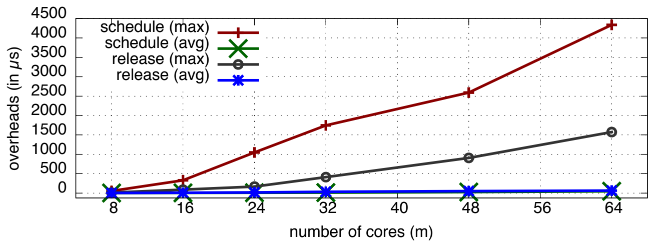


Fig. 2: Scheduling and release overheads under SD. The average scheduling and release overheads coincide at this resolution.

Both operations require a pairwise acquisition of the relevant per-processor locks in order to maintain consistency. To prevent deadlocks, the locks are always acquired in order of increasing processor indices, which may require processors to drop and reacquire their local scheduler lock. To improve blocking time in the average case, the local lock is always dropped in Linux and both source and target locks are reacquired in order.

In summary, the push/pull design underlying the SD scheduler strives to avoid scalability bottlenecks by limiting cache contention through the use of per-processor run queues, and by using fine-grained, per-processor locks. Consequently, our starting assumption was that SD would easily and consistently outperform GSN-EDF. Surprisingly, we found that the push/pull design works both really well and not at all: as shown in Fig. 2, it yields very low average-case overheads, but it is also subject to excessive worst-case overheads.

On the positive side, our experiments show that the more sophisticated locking mechanism is very useful for coping with contention in the average case. We found the average-case overheads to remain low throughout the tested range, and well below 100 μs even for $m = 64$. From a throughput perspective, SD thus performs excellently, and much better than GSN-EDF.

However, from a (hard) real-time perspective, the results are troublesome, as the maximum observed overheads clearly do not scale well. In fact, in some configurations, SD exhibits maximum overheads more than five times worse than the much simpler GSN-EDF: for instance, for $m = 24$, the maximum scheduling overhead of GSN-EDF is roughly 180 μs, whereas SD already exceeds 1,000 μs. In the extreme of $m = 64$, we even observed scheduling overheads approaching 4.5 ms, which demonstrates a total lack of scalability w.r.t. worst-case overheads.

D. Root Causes: Transitive Blocking and Push Failures

As we set out to investigate the scalability of G-EDF implementations, our initial assumption was that per-processor state and locks as employed by SD would "naturally" be the most promising approach, a fallacy that we believe to be widespread. In fact, to the best of our knowledge, our experiments are the first to demonstrate that, w.r.t. maximum overheads, the push/pull approach scales significantly worse than a single, coarse-grained lock. In the end, it turns out that "simulating" a unified global state from multiple distributed states requires complex synchronization logic (*e.g.*, scanning operations, the release and reacquisition of locks, complex interaction among processors, *etc.*), which is detrimental to worst-case overheads.

A key reason for the observed spikes is that fine-grained locking gives rise to pathological, difficult to anticipate blocking scenarios when multiple processors push and pull simultaneously. While the per-processor locks are *on average* accessed by only a handful of processors, it is *possible* for any lock to be contested by all processors at the same time. Further, due to the scanning operations and the need for pairwise ordered lock acquisitions described above, this effect can escalate due to *transitive blocking* when high-contention scenarios are encountered on every outer and every nested lock acquisition.

The case for push/pull-based designs is further weakened by an "optimization" inherited from stock Linux's push/pull scheduler that prevents SD from maintaining the global invariant at all times (*i.e.*, priority inversions, though likely infrequent, are possible with the current implementation). The problem arises because a push operation may fail as follows. To carry out a push migration, the source processor first searches for the processor that is currently executing the lowest-priority task (*i.e.*, earliest-deadline task in SD) by iteratively examining all processor states. After the target processor has been identified, its lock is reacquired to enact the migration. However, in the mean time while the source processor was not holding the target processor's lock, the target processor's state may have changed, thus invalidating the decision. Specifically, after deciding to migrate a task T_i to a processor P_k, a new, higher-priority task may arrive on P_k, either because it resumes or because another processor pushes first (as described in further detail in [13, Ch. 3] in the context of the stock Linux fixed-priority push/pull scheduler). This poses a fundamental question: in a push/pull design, how often must a processor retry a failed push operation in the worst case?

In the current SD implementation (version V8), if a push operation fails three times, then it is abandoned and the task is kept in the local queue even if that violates the global scheduling invariant. However, a retry limit of three is clearly insufficient, since with tens of cores and hundreds of tasks, it is certainly possible that more than three higher-priority tasks arrive concurrently with a push operation. Unfortunately, the hard-coded retry limit exists for a reason: correcting it would likely cause a further increase in (worst-case) overheads due to additional contention from repeatedly failing push operations.

Finally, it should be noted that overheads were measured while executing simple, periodic, CPU-bound, synchronously[2] released real-time tasks, as explained in Appendix A. The

[2]In a *synchronous* task set release, all tasks release their first job at the same time, which ensures that all tasks share a common time zero. In our case, the synchronous release serves to provoke worst-case contention and overheads.

synthetic benchmark tasks used to evaluate SD correspond to the standard workload used to benchmark LITMUS$^{\mathrm{RT}}$ (see Sec. IV) and mimic well-behaved recurrent real-time applications such as process control or signal processing tasks. We thus believe this to be a fair and relevant assessment of the two G-EDF implementations, but note that "antagonistic" tasks could likely provoke even higher overheads under either scheduler. Nonetheless, the results allow us to identify the central scalability bottleneck.

E. Scalability Implications

In the end, we found that neither fine-grained nor coarse-grained locking scales particularly well. This is because the true bottleneck is not *lock granularity*, which is more relevant in a throughput context, but rather *peak contention*, that is, the number of processors that potentially access the *same* lock(s). Peak contention, however, cannot be reduced by splitting the shared global state into ever finer pieces that still must be shared among all processors. To the contrary, due to transitive blocking, m processors sharing m locks in a nested fashion suffer larger worst-case delays than m processors sharing a single lock.

Ideally, a scheduler should have the simplicity of GSN-EDF, provide average-case overheads close to those of SD, and provide worst-case overheads as predictable and simple to reason about as those of GSN-EDF (but preferably lower). In the following, we present a novel G-EDF implementation, based on message passing instead of state sharing, that comes much closer to these goals than any prior implementation known to us.

III. Global Scheduling via Message Passing

The use of message passing is a growing trend in recent work on multicore scalability [9, 22, 34]. Its main advantage is that it makes sharing explicit and forces algorithm designs that inherently provide data isolation and reduce sharing. In the following, we show that message passing is also a viable implementation technique at the timescale of a real-time scheduler. We begin with an overview of G-EDF-MP's design, our new message-passing-based G-EDF implementation, and provide a detailed description of how it operates in Sec. III-B.

A. Design Overview

As discussed in Sec. II, our solution is motivated by the fact that large overheads from peak contention and cache-line sharing arise because each processor must coordinate with other processors to generate a globally consistent snapshot of the system prior to making a decision.

G-EDF-MP avoids these two issues through a simple tradeoff. Taking dedicated interrupt handling as already implemented in GSN-EDF a significant step further, it declares a single *dedicated scheduling processor* (henceforth the DSP) to be responsible for all scheduling decisions and all device and timer interrupts. The rest of the *client* processors in the system simply carry out the decision made by the DSP. To avoid analytical complications, the DSP itself does not execute any real-time tasks, which execute only on the shielded client processors.

Since only the DSP makes decisions, it is the only processor that needs to have a global view of the system. G-EDF-MP thus maintains this view locally in the DSP: ready queue updates do

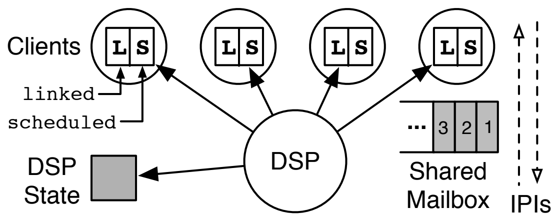


Fig. 3: Structural overview of G-EDF-MP.

not require any locking, to the effect that job release interrupts cannot be delayed by lock contention.

However, for reasons of efficiency, G-EDF-MP is not a "pure" message-passing-based implementation, as it still uses some locks to synchronize the assignment of tasks to clients. In particular, the DSP-to-client communication is carried out with two lock-protected task pointers per client, which the DSP updates in order to indicate task assignments. However, the peak contention of these locks is limited because **(i)** they are never acquired in a nested fashion, and **(ii)** they are subject to only pairwise sharing (by the DSP and the particular client).

The DSP uses *inter-processor interrupts* (IPIs) to notify clients when a particular decision requires an immediate preemption. Similarly, a client communicates local scheduling events (*e.g.* job completions, suspensions, *etc.*) to the DSP by writing a message to one of a small number of shared, wait-free FIFO mailboxes and sending an IPI as a notification. This allows the DSP to keep track of the global state of the system and to react as clients become available for other work.

Overall, the G-EDF-MP approach is conceptually simple, but requires careful implementation choices, which we discuss next.

B. The G-EDF-MP Data Structures

Fig. 3 provides an overview of the system. The DSP uses its local *DSP state* to make scheduling decisions. The clients have a small *per-client* state that is protected by per-client locks (not shown), which indicates the assigned and scheduled tasks. Preemptions are enacted by the DSP by modifying a client's state and sending an IPI. Clients communicate with the DSP by writing to a *shared mailbox* and sending an IPI to the DSP.

Per-client state. G-EDF-MP is conceptually based on the link-based scheduling approach first introduced with GSN-EDF [17].

Link-based scheduling tracks two task pointers for each processor, the `linked` task that *should be* scheduled, and the `scheduled` task that currently *is* scheduled. The DSP updates the `linked` task as required to maintain the global scheduling invariant, and the client is responsible for tracking `linked` with `scheduled` as quickly as possible by context-switching to `linked` whenever the DSP updates the assignment. (This delay is accounted for analytically in Appendix B.)

The two main advantages of link-based scheduling are that it enables the implementation of non-preemptable critical section support with $O(1)$ priority inversion [13, 15] (though this is not yet activated in G-EDF-MP), and that it cleanly decouples the logical scheduling aspects as dictated by the scheduling policy from the peculiarities of hardware and process management [13].

Each client's `linked` and `scheduled` fields are protected by a spin lock. Only clients may change their `scheduled`

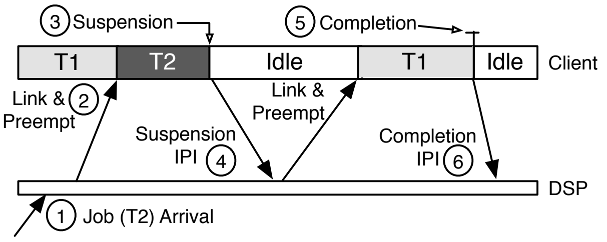


Fig. 4: Overview of G-EDF-MP operations.

pointer, but both the DSP and the clients update the `linked` pointers: clients invalidate their `linked` pointer to indicate job completions or suspensions, and the DSP updates the `linked` pointer to assign new work. In contrast, under GSN-EDF, all processors may relink any other processor and all `linked` and `scheduled` pointers are protected by a single lock.

DSP state. The DSP state consists of **(i)** a ready queue of tasks that are ready to execute but are waiting to be linked, **(ii)** binomial heaps and timers tracking future job releases,[3] and **(iii)** a copy of each client's state (*e.g.* which task is linked to, whether it is idle *etc.*) that is updated whenever the DSP changes `linked` or when a client reports idleness. Part (iii) is used to detect when preemptions become necessary. The DSP uses a copy of the client state (rather than accessing it directly) to improve cache locality and to avoid having to lock per-client states during preemption checks, which are frequently required. Notably, the DSP state is not protected by any locks since it is accessed only by the DSP with disabled interrupts.

Shared mailbox. The mailbox enables asynchronous client-to-DSP communication, where each message consists of a pointer to a callback function, the sender ID, and the task that triggered the scheduling event. It consists of a small number of Feather-Trace buffers [17], which support wait-free writes so clients never block when sending messages.[4] To take advantage of the cache topology, there is one mailbox per socket that is shared only by cores on the same die (and the DSP). The DSP services message buffers in a round-robin fashion to avoid starvation, and briefly enables interrupts between messages (lest latency spikes arise). We next describe the main scheduling events.

C. *The G-EDF-MP Operations*

There are six major message types in G-EDF-MP, as illustrated in Fig. 4, which shows a sequence of events and messages.

1) When a job is released or resumed by an external event, the DSP receives an interrupt (since we use dedicated interrupt handling) and the associated handler is executed. Otherwise, if a task is woken up by another task on a client, the DSP is notified by the client via a message. In either case, the DSP immediately links the task to an idle core if one is available, and otherwise checks whether a preemption is in order. Since the DSP has a copy of all current task assignments, these decisions are made purely locally without any lock acquisitions.

In Fig. 4, the DSP decides that the newly arrived task T_2 should preempt task T_1 on the depicted client processor. To enact the preemption, the DSP acquires the client's lock, updates its `linked` pointer, and sends an IPI. After releasing the client's lock, the DSP adds the unlinked task T_1 back into the ready queue. Note that T_1 remains scheduled until the client receives the IPI, which illustrates the separation of logical and physical processor management in link-based scheduling.

2) When a client processor receives a rescheduling IPI, it locks its local state and simply checks whether `linked` differs from `scheduled`, and if so atomically updates `scheduled` to indicate that it has picked up the link change. It then unlocks its local state and carries out a context switch to the new task.

3) When a job suspends on a client processor, it immediately becomes unavailable for execution. Hence, the client invalidates its `linked` field to indicate that it is idle and sends a message to the DSP to communicate its availability. Since the client at this point does not have information about which job to schedule next (or even whether more jobs are pending), it simply idles (or services non-real-time background tasks).

Introducing idle time of course introduces additional delays for any pending real-time tasks, which must be reflected when accounting for overheads, as we discuss in Appendix B. Note that this idle time is not in conflict with the fact that G-EDF is a work-conserving policy. In any practical scheduler implementation, processors incur brief periods of unavailability while selecting the next task to dispatch, regardless of whether this is realized with locking or message passing mechanisms.

4) When the DSP is notified of a suspension, it discards the suspending task from its data structures. The suspending task will be "reintroduced" to the scheduler when it is later resumed by an interrupt or another task—see (1) above. Note that a suspending task might already be queued in the ready queue if a preemption IPI sent from the DSP to the client raced with a suspension IPI sent from the client to the DSP. In general, since G-EDF-MP intentionally aims to decouple clients from the DSP as much as possible for scalability reasons, many potential race conditions among in-flight events must be handled.

Since a suspension message implies that the sending client became idle, the DSP links the highest-priority job in the ready queue to the sender (if any). In the example in Fig. 4, the DSP links T_1 to the client and sends a preemption IPI, which causes T_1 to be dispatched as discussed in (2) above.

5) When a job completes on a client processor, the client reacts as in (3): it invalidates `linked` and notifies the DSP.

6) When the DSP is notified of a completion, if the job completed before its next release time, the DSP programs a local timer for the next job release; otherwise, if the job was tardy, it adds the task immediately back into the ready queue. Either way, it selects the highest-priority job in the ready queue (if any) and assigns it to the sender.

Concurrency challenges. The preceding list of events covers the major scheduling operations in G-EDF-MP. While the design

[3]LITMUS^{RT} provides binomial heaps to enable the release of many jobs at once with efficient $O(\log n)$ merge operations [18]. G-EDF-MP simply reuses the existing infrastructure, with the difference that no locking is required.

[4]We chose Feather-Trace buffers because they are readily available in LITMUS^{RT} as part of the overhead tracing infrastructure. In future work, it may be interesting to evaluate whether specific-purpose, single-writer, cache-coherency-friendly message buffers [9, 22] achieve even lower overheads.

is intentionally simple at the conceptual level, several corner cases must be correctly dealt with in practice. For example, since an unlinked task T_i in the ready queue may remain briefly scheduled on some processor P_k (until the client's "is"-state catches up with the "should"-state), it may be picked by the DSP and linked to some other processor P_l that scheduled some other task T_j. In such a case, P_l must delay its context switch from T_j to T_i until P_k ceases to use T_i's kernel stack.

This, however, becomes problematic if subsequently T_j is assigned to P_k—a deadlock results, as P_l uses T_j's kernel stack while it waits for P_k to release T_i's kernel stack, and *vice versa*. If left unchecked, this can happen as a result of short suspensions (*e.g.*, due to I/O), and must be detected and resolved by swapping the links of T_i and T_j. To avoid such situations altogether, G-EDF-MP performs link swaps whenever it links a still-scheduled task. To allow the DSP to reliably detect still-scheduled tasks, clients must check `linked` and update `scheduled` atomically, as discussed in operation (2).

While such corner cases add some implementation complexity, they do not affect the overheads and scalability of G-EDF-MP negatively, as we show next.

IV. Evaluation

To assess the scalability of G-EDF-MP, SD, and GSN-EDF in terms of average and maximum overheads, we conducted extensive experiments on a 64-core Intel Xeon X7550 2.0 GHz platform using LITMUS$^{\mathrm{RT}}$ version 2013.1 and version V8 of the SD patch, both based on Linux 3.10. Features that lead to unpredictability such as hardware multithreading, frequency scaling, and deep sleep states were disabled for all kernels, along with every kernel configuration option associated with debugging. All non-essential background services such as cron were disabled. For each $m \in \{8, 16, 24, 32, 48, 64\}$, each of the three kernel versions was compiled with support for at most m processors and booted on the test platform.

Overheads were measured with Feather-Trace [17], a lightweight tracing framework included in LITMUS$^{\mathrm{RT}}$. As Feather-Trace was not built with large core counts in mind, the stock version uses a single, shared trace buffer, which itself creates a scalability bottleneck at higher core counts. To resolve this, we changed Feather-Trace to record most samples in processor-local buffers. We also added support for coping with non-synchronized cycle counters by determining each processor's cycle counter offset prior to each experiment.

For each m and for each $n \in \{m, 2m, \ldots, 10m\}$, we generated five task sets with n synthetic tasks using Emberson *et al.*'s task set generator [25]. Task sets were generated with a target utilization chosen uniformly at random from $[0.4m, 0.8m]$ and periods drawn from a log-uniform distribution in the range $[10ms, 100ms]$. Each task set was executed for 30 s under each of the following four schedulers: G-EDF-MP, SD, GSN-EDF *without* dedicated interrupt handling, and GSN-EDF *with* dedicated interrupt handling (GSN-EDF-DI hereafter). Note that real-time tasks execute on only $m - 1$ of the m cores under G-EDF-MP and GSN-EDF-DI. We ran a cache-thrashing background workload to provoke maximum memory hierarchy

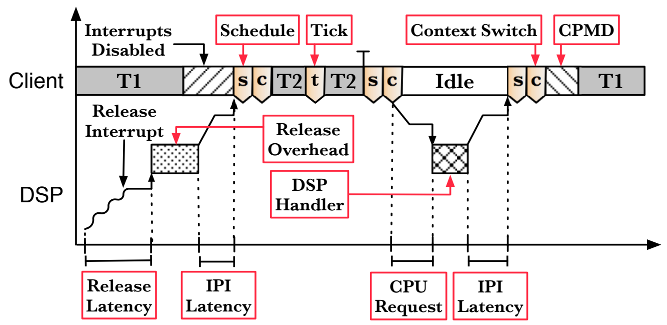


Fig. 5: Overview of G-EDF-MP runtime overheads.

contention. In total, we traced 1,200 task sets during ten hours of real-time execution and collected over 7 billion raw samples.

Next, we discuss the different runtime overheads that delay real-time tasks, which are all traced individually in LITMUS$^{\mathrm{RT}}$.

A. Measured Runtime Overheads

In the case of G-EDF-MP, nine major overhead sources affect a task's response time, which are illustrated in Fig. 5.

First, a task incurs *release event latency* (Δ^{ev}) if its triggering interrupt is delayed because the DSP disabled interrupts. *Release overhead* (Δ^{rel}) reflects the cost of processing the release interrupt, including all ready queue updates, preemption checks, and link changes. After the link change, the DSP sends a rescheduling IPI, which incurs *IPI latency* (Δ^{ipi}) due to interconnect contention and when the recipient disabled interrupts.

On the client, the task incurs *scheduling overhead* (Δ^{sch}) while the kernel selects the next task to be dispatched; this overhead includes any blocking delays incurred when accessing the scheduler state. Finally, *context-switch overhead* (Δ^{cxs}) is incurred when switching kernel stacks. While executing, a real-time task can be delayed by the periodic timer tick (which fires once every millisecond), incurring *tick overhead*.

Two overheads specific to G-EDF-MP are the IPI sent to notify the DSP of pending messages and the time spent by the DSP reacting to messages. Regarding the former, we distinguish between Δ^{ipi}, the latency of normal reschedule IPIs sent to trigger the scheduler on a remote processor, and *client request latency* (Δ^{req}), which reflects the delay from the time that a client sends a message to the DSP until the DSP dequeues that specific message. Regarding the latter G-EDF-MP-specific overhead, each message has a corresponding callback function, the invocation of which causes *DSP handler overhead* (Δ^{dsp}).

CPMD (Δ^{cpd}) affects a task when it reloads cache contents after a preemption or migration. CPMD is not affected by the choice of G-EDF implementation and was measured separately using the method proposed by Bastoni *et al.* [8].

With the exception of the client request latency and DSP handler overhead, all of the just-discussed overhead sources also affect GSN-EDF, GSN-EDF-DI, and SD. To obtain comparable measures for SD, we ported Feather-Trace to the SD kernel (but did not introduce any other changes). Further, the scheduling overhead Δ^{sch} is actually measured in two parts (before and after the context switch); we report the aggregate measure here. In contrast to earlier LITMUS$^{\mathrm{RT}}$-based studies [7, 16, 18], we did not apply any statistical outlier filters (which made possible by recent LITMUS$^{\text{RT}}$ improvements [14]). Finally, it is also worth mentioning that synchronously releasing all tasks (as discussed in Sec. II-D and Appendix A) creates a challenging benchmark, as it forces all real-time tasks to be activated together. The overhead measurements that were obtained thus reflect very unfavorable scenarios in terms of contention.

All graphs are available online [21]. Due to space constraints, we discuss only the most pertinent trends in the following.

B. Experimental Results

As already shown in Figs. 1 and 2, GSN-EDF does not scale well w.r.t. both average- and worst-case overheads, and SD does not scale well w.r.t. worst-case overheads. The key questions thus are, **(i)** does G-EDF-MP scale well with the number of cores and the number of tasks, and **(ii)** does G-EDF-MP achieve low average-case overheads similar to SD?

We begin with question (i), and the answer is clearly *yes*. Fig. 6(a) shows the maximum observed scheduling overhead as a function of m. While the maximum scheduling overhead under GSN-EDF and SD increases from 40.19 μs and 52.11 μs, resp., at 8 cores to 2759.04 μs and 4337.55 μs, resp., at 64 cores, the scheduling overhead under G-EDF-MP increases from 12.75 μs to only 119.37 μs; a relatively modest increase and a 23x and 36x improvement over GSN-EDF and SD, resp.

Similar trends are observed with increasing n, as shown in Fig. 6(c), which depicts Δ^{sch} as a function of n for $m = 48$. Whereas the maximum scheduling overhead under GSN-EDF, GSN-EDF-DI, and SD either already start high or reveal large upwards slopes, G-EDF-MP maintains near-constant overheads around 60 μs for the entire range of tested task set sizes. These trends are not surprising, since GSN-EDF, GSN-EDF-DI, and SD must acquire one or more globally shared locks (for which contention increases with m) and manipulate ready queues (which grow in size with n) as part of each scheduler invocation. Further, overheads under SD are strongly affected by increasing n since a larger number of tasks increases the frequency of scheduler invocations, which increases the likelihood of push failures and transitive blocking. In contrast, each client state lock in G-EDF-MP is accessed by only two processors, and clients never access the ready queue—the client-side scheduler is $O(1)$.

GSN-EDF and GSN-EDF-DI exhibit mostly similar trends, with overheads slightly lower under GSN-EDF-DI simply because one fewer core participates in the scheduling, thereby reducing peak contention slightly.

The major difference in scalability between the primarily lock-based schedulers and G-EDF-MP also manifests in the maximum release overhead as shown in Fig. 6(b). The overhead under G-EDF-MP exhibits quicker growth than in Fig. 6(a) due to $O(\log n)$ updates to the ready queue (a binomial heap) and $O(m)$ preemption checks. Still, G-EDF-MP exhibits much lower maximum release overhead than its counterparts afflicted by lock contention: for $m = 64$, the maximum release overhead exceeds 1700 μs and 1500 μs under GSN-EDF and SD, resp., whereas G-EDF-MP stays well below 275 μs.

To answer question (ii), consider the plots of average-case scheduling and release overheads shown in Figs. 6(d) and 6(e), resp. As discussed in Sec. II, GSN-EDF does not scale well in the average case, which also extends to GSN-EDF-DI, whereas SD has no problems scaling in the average case (since pathological contention is rare). G-EDF-MP in turn exhibits still lower scheduling overheads, as seen in Fig. 6(d) (at $m = 64$, $\Delta^{sch} = 3.04\,\mu s$ under G-EDF-MP vs. $\Delta^{sch} = 48.48\,\mu s$ under SD).

In Fig. 6(e), G-EDF-MP exhibits somewhat higher average release overheads than SD (at $m = 64$, $\Delta^{rel} = 109.40\,\mu s$ under G-EDF-MP vs. $\Delta^{rel} = 65.67\,\mu s$ under SD), but more importantly, the overhead under G-EDF-MP grows only slowly from $m = 8$ to $m = 64$, in contrast to GSN-EDF (which is below 6 μs at $m = 8$, but exceeds 475 μs at $m = 64$).[5] We conclude that G-EDF-MP already performs reasonably well in terms of average-case overheads, but also believe that there is room for further low-level code optimizations in the future.

Although the trends are promising so far, there is no silver bullet for multicore scalability—the design of G-EDF-MP introduces new overheads that also need to be considered. These are shown in Fig. 6(f), which depicts the average and maximum client request latency and the DSP handler cost.

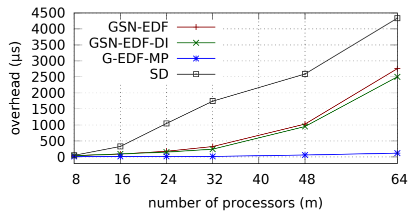
The DSP handler cost is not particularly problematic since it is dominated by ready queue and preemption checks and thus grows only slowly with m, just like the release overhead.

The client request latency is determined by three factors: how long the DSP disables interrupts, how expensive it is for the DSP to dequeue a message, and how many messages are already queued (recall that each message buffer is FIFO-ordered). To keep the first factor low, the DSP enables interrupts between any two messages. Concerning the second factor, reading the mailbox becomes more expensive with growing core counts as more processors send messages concurrently. However, since we use per-socket mailboxes, this increase is relatively small.

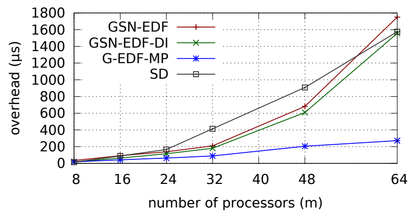
The third factor, the length of the message queue, is the main bottleneck of G-EDF-MP—with an increasing number of clients, the peak contention in terms of simultaneous message arrivals naturally grows as well. This can be clearly seen in Fig. 6(f), where the maximum client request latency grows quickly with m. The average client request latency, however, is not strongly affected by this—again, peak contention is a rare event.

Finally, to offer further support for our claims and to illustrate how overheads vary with increasing task counts, Fig. 7 depicts six additional graphs showing scheduling overheads, release overheads, DSP handler overheads, and client request latency as a function of n on the 32-core platform. The general trend is that G-EDF-MP performs well w.r.t. the overheads encountered by all schedulers—in insets (a), (b), (d), and (e), G-EDF-MP exhibits the lowest overheads of the four schedulers for any n. In fact, in special cases, G-EDF-MP achieves even up to two orders of magnitude lower overheads than the reference implementations: the maximum observed scheduling overhead under G-EDF-MP is $\approx$17 μs vs. $\approx$1740 μs under SD, as apparent in Fig. 7(a). Finally, Fig. 7(c) confirms that client request latency is the main bottleneck w.r.t. worst-case overheads, but not a major problem w.r.t. average-case overheads, and Fig. 7(f) shows that the cost

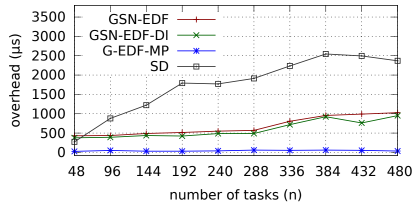
[5]Release operations are costly in both GSN-EDF and GSN-EDF-DI because, in the worst case, all tasks are released simultaneously and every processor must be preempted, which requires many updates of the global state and results in significant lock and cache contention.



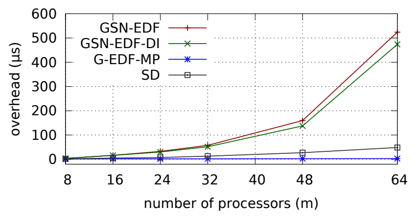
(a) Maximum scheduling overhead ($m \in [8, 64]$)



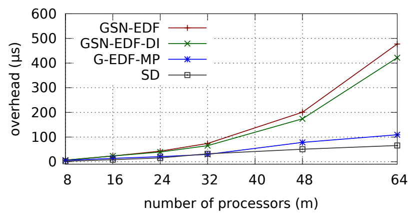
(b) Maximum release overhead ($m \in [8, 64]$)



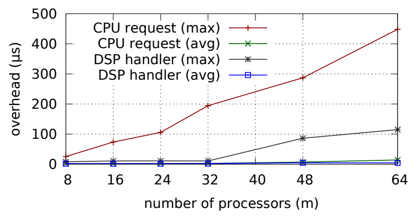
(c) Maximum scheduling overhead ($m = 48$)



(d) Average scheduling overhead ($m \in [8, 64]$)

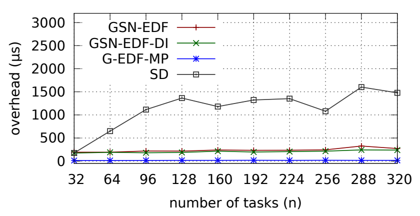


(e) Average release overhead ($m \in [8, 64]$)

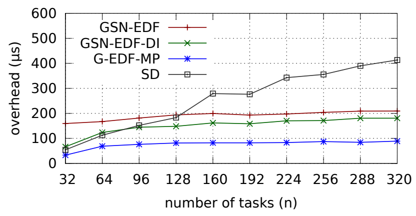


(f) Average & maximum G-EDF-MP overheads ($m \in [8, 64]$)

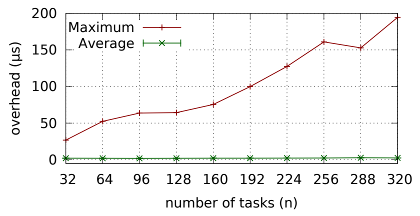
Fig. 6: Runtime overheads under GSN-EDF, GSN-EDF-DI, SD, and G-EDF-MP. Note that each graph uses a different scale due to the wide range of observed overhead magnitudes.



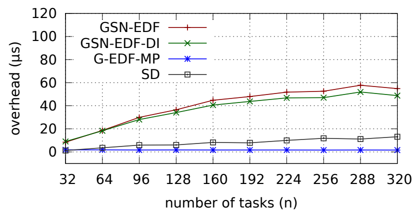
(a) Maximum scheduling overhead ($m = 32$)



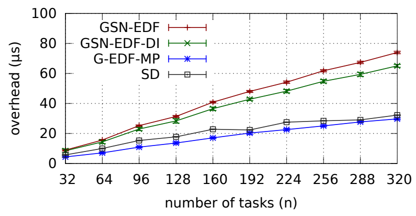
(b) Maximum release overhead ($m = 32$)



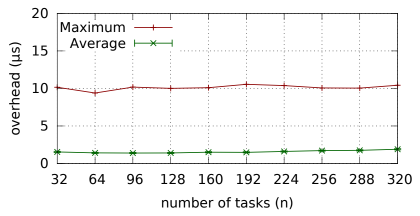
(c) Client request latency ($m = 32$)



(d) Average scheduling overhead ($m = 32$)



(e) Average release overhead ($m = 32$)



(f) DSP handler overhead ($m = 32$)

Fig. 7: Additional results showing overheads as a function of increasing n for $m = 32$. Note that each graph uses a different scale due to the wide range of observed overhead magnitudes.

per DSP operation is virtually constant.

Overall, our data shows G-EDF-MP to scale much better than either GSN-EDF or SD in the tested range of up to 64 cores. Nonetheless, G-EDF-MP is obviously not arbitrarily scalable as the DSP eventually becomes a bottleneck, albeit at much higher processor counts than in prior lock-based schedulers.

Next, we discuss whether G-EDF-MP's improved overhead scalability translates into analytical benefits as well.

V. Schedulability Experiments

Since message passing introduces new sources of delays, just comparing the magnitude of overheads does not provide a complete view of G-EDF-MP's performance. Rather, *overhead-aware* schedulability analysis that accounts for all runtime overheads is necessary. We therefore performed extensive overhead-aware schedulability experiments to assess the impact of overheads on each of the four implementations. Before presenting the main results, we briefly summarize our experimental setup.

A. Experimental Setup

We evaluated the four schedulers (GSN-EDF, GSN-EDF-DI, SD and G-EDF-MP) w.r.t. two criteria: *soft* real-time schedulability, by checking if a task set has *bounded tardiness* [24] after accounting for *average-case* overheads, and *hard* real-time schedulability, by applying a collection of G-EDF schedulability tests [2, 5, 10, 28] after accounting for *worst-case* overheads. Runtime overheads were accounted for using standard techniques as described in Appendix B. For increased breadth, we employed two different task set generation methods; one used in prior LITMUS$^{\text{RT}}$ studies [13, 16, 18] and one proposed by

Emberson *et al.* [25].

First, we evaluated schedulability as a function of n using task sets similar to those used in previous LITMUS$^{\mathrm{RT}}$ studies (see [13, Ch. 4] for a detailed description). In short, each task set is generated by independently drawing n utilizations and periods. Utilizations were drawn from one of six distributions: three exponential distributions ranging over $[0, 1]$ (with means 0.1, 0.25 and 0.5), and three uniform distributions (ranging over $[0.1, 0.2]$, $[0.1, 0.4]$ and $[0.5, 0.9]$). Periods were chosen from uniform and log-uniform distributions across several ranges ($[1, 1000]$, $[10, 100]$, $[10, 1000]$, $[3, 33]$, and $[200, 1000]$, in ms).

In the second set of experiments using Emberson *et al.*'s task set generation method [25], we assessed the impact of increasing the total utilization for a fixed n. We varied $n \in \{2m, 3m, \ldots, 10m\}$ and considered both uniformly and log-uniformly distributed periods in the range $[1, 1000]$.

Both methods were used to assess the schedulability for each $m \in \{8, 16, 24, 32, 48, 64\}$. For each sampling point, we generated and tested 1,000 task sets.

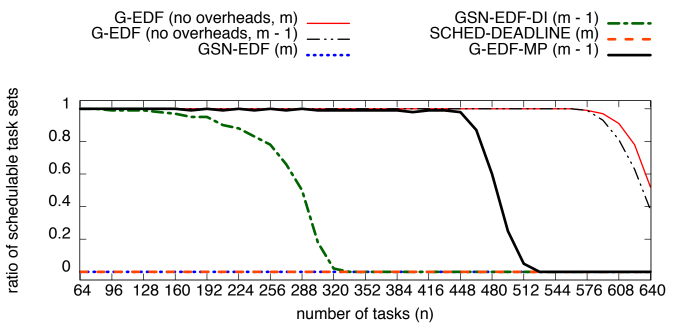
In total, we evaluated more than 1,000 parameter configurations. Due to space constraints, we highlight here two configurations that reveal a significant analytical impact. The complete set of results is available online [21].

B. Results

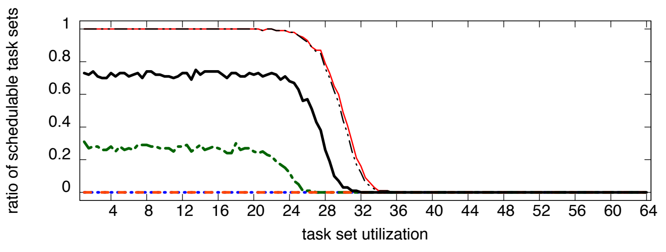
Fig. 8(a) shows a result depicting soft real-time schedulability for $m = 64$ as a function of n. The achieved soft real-time schedulability is shown for each of the four evaluated implementations and also for two hypothetical G-EDF configurations assuming zero overheads on m and $m - 1$ processors, to provide an upper bound and to show the cost of using a DSP. In Fig. 8(a), GSN-EDF and SD achieve zero schedulability for all task counts due to the pessimism involved in interrupt accounting (recall that GSN-EDF-DI and G-EDF-MP shield real-time tasks from all interrupts, whereas tasks may be delayed by release interrupts in SD and GSN-EDF). Under G-EDF-MP, bounded tardiness is maintained up to $n = 448$, whereas GSN-EDF-DI starts to decay already at $n = 128$ and reaches zero schedulability at $n = 320$. Here, G-EDF-MP can support more than 120 additional tasks, thereby halving the distance to the theoretical upper bound.

Fig. 8(b) shows hard real-time schedulability as a function of total utilization (for a fixed n). As in Fig. 8(a), interrupt-related pessimism renders GSN-EDF and SD completely unschedulable. About one third of the task sets are schedulable under GSN-EDF-DI until reaching a total utilization of about 20. In contrast, about 70% of the task sets are schedulable under G-EDF-MP until reaching a total utilization of about 24, where the theoretical upper bound starts to decrease as well. Again, G-EDF-MP halves the gap to the theoretical upper bound.

Fig. 8(b) also exhibits G-EDF's inherent hard real-time limitations: even when assuming overhead-free execution, hardly any task set with a total utilization greater than 50% can be claimed schedulable. However, in the context of this work, G-EDF's *algorithmic* limitations are irrelevant because G-EDF-MP's low-overhead design can be easily transferred to other global schedulers (such as global fixed-priority scheduling or



(a) Soft schedulability as a function of n for exponentially distributed task utilizations with mean 0.1 (as in [13, 16, 18]).



(b) Hard schedulability as a function of total utilization based on Emberson *et al.*'s method [25] with $n = 5m$.

Fig. 8: Comparison of hard and soft schedulability for $m = 64$. Periods were drawn from a log-uniform distribution with range [$1ms$,$1000ms$].

earliest-deadline-until-zero-laxity scheduling (EDZL) [3], which can offer higher hard real-time schedulability).

Overall, the schedulability experiments confirm that G-EDF-MP's lower overheads translate into significantly higher schedulability at high core counts. At lower core counts (*i.e.*, $m \leq 16$) and with fewer tasks, overhead scalability is not yet an issue and G-EDF-MP performs similarly to GSN-EDF-DI. The complete set of experiments is available online [21].

VI. Related Work

This work reuses two key implementation techniques and is closely related to several prior studies, which we review next.

Implementation techniques. G-EDF-MP employs a DSP to shield real-time tasks from OS overheads, a well-known technique that dates back to the Spring kernel [38]. More recently, the idea of delegating cores to exclusively handle OS duties has also been applied in the context of manycore platforms (*e.g.*, [4, 39]).

G-EDF-MP further employs message passing, which is known to scale well on multicores. In particular, in *multikernel* OSs such as Barrelfish [9] and Quest-V [34], the entire OS design is centered around message passing to avoid scalability bottlenecks (among other reasons). Message passing is further used in some current commercial RTOSs as well (*e.g.*, ENEA's OSE). However, G-EDF-MP differs significantly from these prior works in that it implements a *global* scheduling policy, wheres OSE, Quest-V, and Barrelfish intentionally avoid global policies whenever possible and implement partitioned schedulers (which scale trivially w.r.t. overheads).

Message passing has also been shown to offer higher throughput than shared locks in the face of heavy contention in recent work studying synchronization choices on multicore platforms [22, 35]. However, throughput depends mostly on average-case overheads, whereas we designed G-EDF-MP specifically to lower worst-case overheads. Prior to G-EDF-MP, it was not clear that message passing could be a viable—or even superior—choice for constructing global real-time schedulers.

Prior studies. Both GSN-EDF and SD have been the subject of prior studies. Most recently, Lelli *et al.* [33] empirically compared SD and stock Linux's SCHED_FIFO scheduler (in various configurations to emulate global, clustered, and partitioned scheduling) on a 48-core platform. In contrast to this study, they did not measure worst-case overheads (and did not apply schedulability analysis), but rather measured average-case overheads and indirectly assessed each scheduler's performance with a statistical evaluation of observed task behavior. Since SD exhibits low average-case overheads, their experimental setup did not reveal the pathological lock contention that we observed.

Several alternative implementations of G-EDF in LITMUS^{RT} were compared in [16]. While several of the investigated alternatives aimed at reducing lock contention, they did not perform significantly better than the stock GSN-EDF plugin, which in hindsight can be attributed to the fact that they did not reduce peak contention (a single lock to serialize scheduling decisions remained in all versions). Dedicated interrupt handling was shown to be very beneficial, which is apparent in Fig. 8, too.

GSN-EDF was previously evaluated as part of several large-scale, overhead-aware schedulability studies [7, 13, 18], as was a prior version of G-EDF in LITMUS^{RT} [20]. However, these studies were conducted on platforms significantly smaller than the 64-core machine employed herein, and also considered only a single core count each; scalability limitations hence did not manifest to the extent revealed in this paper. Further, prior studies [7, 13, 18, 20] did not compare LITMUS^{RT} against SD and used statistical outlier filters (in contrast to this study).

Finally, it should be noted that scheduling overheads are just one piece of the predictability puzzle on multicores, as other factors such as contention and interference at all levels of the memory hierarchy must be considered, too (*e.g.*, see [30, 36]).

VII. Conclusion

We evaluated the scalability of G-EDF implementations and found that G-EDF-MP, a novel scheduler design based primarily on message passing, scales substantially better than prior lock-based schedulers, thereby lowering scheduling overheads by up to a factor of 36 on 64 cores. While global scheduling is certainly not the right approach for *all* real-time workloads (*e.g.*, CPMD issues remain), this work significantly extends the range of workloads and platforms for which global approaches are viable, thus opening up interesting avenues for future work. In particular, the presented approach is not limited to G-EDF, but can be easily applied to other global scheduling policies as well.

The key limitation of G-EDF-MP is that a single DSP is an inherent bottleneck that inevitably will cause problems beyond 64 cores. Given this limitation, systems with core counts too large to be efficiently supported by a single G-EDF-MP instance should use clustered scheduling instead (whereby the set of all cores is partitioned into disjoint subsets and a global scheduler is instantiated for each such subset). In fact, this was already noted in a prior study [7], which recommended that global scheduling should be restricted to a "small-to-medium number of cores." Our work does not invalidate this observation, but rather redefines the notion of what constitutes a "medium number of cores." That is, seen in this context, G-EDF-MP simply increases the largest cluster size that can be efficiently supported.

In future work, it would be interesting to develop G-EDF-MP extensions with two or more DSPs, which then, however, would have to synchronize decisions. A substantial algorithmic improvement would be to equip clients with a `next` pointer to avoid idle time after job completions/suspensions, but choosing `next` such that analytical benefits are realized is not trivial.

References

[1] "The LITMUS^{RT} project web site," http://www.litmus-rt.org.

[2] T. Baker and S. Baruah, "An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems," *Real-Time Systems*, vol. 43, no. 1, pp. 3–24, 2009.

[3] T. Baker, M. Cirinei, and M. Bertogna, "EDZL scheduling analysis," *Real-Time Systems*, vol. 40, no. 3, pp. 264–289, 2008.

[4] F. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador, "Nix: a case for a manycore system for cloud computing," *Bell Labs Tech. Journal*, vol. 17, no. 2, pp. 41–54, 2012.

[5] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *RTSS'07*, 2007.

[6] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *IPPS'95*, 1995.

[7] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *RTSS'10*, 2010, pp. 14–24.

[8] ——, "Cache-related preemption and migration delays: Empirical approximation and impact on schedulability," in *OSPERT'10*, 2010.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *SOSP'09*, 2009.

[10] M. Bertogna, M. Cirinei, and G. Lipari, "Improved schedulability analysis of EDF on multiprocessor platforms," in *ECRTS'05*, 2005.

[11] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA'07*, 2007.

[12] A. Block, "Adaptive multiprocessor real-time systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2008.

[13] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, The University of North Carolina at Chapel Hill, 2011.

[14] ——, "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling," in *RTAS'13*, 2013, pp. 292–302.

[15] B. Brandenburg and J. Anderson, "Feather-trace: A light-weight event tracing toolkit," in *OSPERT'07*, 2007.

[16] ——, "On the implementation of global real-time schedulers," in *RTSS'09*, 2009.

[17] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson, "LITMUS^{RT}: A status report," in *RTLWS'07*, 2007.

[18] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *RTSS'08*, 2008.

[19] B. Brandenburg, H. Leontyev, and J. Anderson, "An overview of interrupt accounting techniques for multiprocessor real-time systems," *Journal of Systems Architecture: Embedded Software Design*, vol. 57, no. 6, pp. 638–654, 2011.

[20] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS'06*, 2006.

[21] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing (extended version)," 2014, available at: http://www.mpi-sws.org/~bbb/papers.

[22] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *SOSP'13*, 2013.

[23] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[24] U. Devi and J. Anderson, "Tardiness bounds under global EDF scheduling on a multiprocessor," *Real-Time Systems*, vol. 38, no. 2, pp. 133–189, 2008.

[25] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," *WATERS'10*, 2010.

[26] J. Erickson and J. Anderson, "Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling," in *ECRTS'12*, 2012.

[27] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, "An EDF scheduling class for the Linux kernel," in *RTLWS'09*, 2009.

[28] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-Time Systems*, vol. 25, no. 2-3, pp. 187–205, 2003.

[29] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities," in *ECRTS'13*, 2013.

[30] Y. Heechul, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS'13*, 2013.

[31] V. Legout, M. Jan, and L. Pautet, "A scheduling algorithm to reduce the static energy consumption of multiprocessor real-time systems," in *RTNS'13*, 2013.

[32] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta, "An efficient and scalable implementation of global EDF in Linux," in *OSPERT'11*, 2011.

[33] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2405–2416, 2012.

[34] Y. Li, M. Danish, and R. West, "Quest-V: A virtualized multikernel for high-confidence systems," 2011, arXiv:1112.5136.

[35] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *ATC'12*, 2012.

[36] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS'13*, 2013.

[37] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *RTSS'11*, 2011.

[38] J. Stankovic and K. Ramamritham, "The Spring kernel: a new paradigm for real-time operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 3, pp. 54–71, Jul. 1989.

[39] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

Appendix

A. Overhead Experiments

Overheads under LITMUS$^{\mathrm{RT}}$ are commonly measured using workloads consisting of synthetic periodic tasks with randomly generated task parameters (*i.e.*, execution costs and periods) [13, 16, 18]. These benchmark tasks carry out floating point arithmetic on large arrays to simulate CPU-bound signal processing tasks. We adopted this approach in this study as well, but needed to port the tasks to SD to enable a fair comparison.

LITMUS$^{\mathrm{RT}}$ provides custom system calls to support periodic job releases and synchronous task set releases (which means that all tasks release their first job at exactly the same time), which we emulated under SD using standard Linux APIs. Periodic execution can be trivially accomplished under Linux with `clock_nanosleep()`. Synchronous task set releases were coordinated with a shared memory segment and *pthreads* barriers, as illustrated in Fig. 9. The benchmark workload hence behaves virtually identically under SD and LITMUS$^{\mathrm{RT}}$.

```
barrier_wait();    // await synchronous release
next_rel = start;  // the shared release time
while (!timeout()) {
    clock_nanosleep(CLOCK_MONOTONIC,
        TIMER_ABSTIME,     &next_rel, NULL);
    // Do some math and dirty the cache...
    next_rel = timespec_add(next_rel,
        period_timespec);
}
```

Fig. 9: Skeleton of the periodic tasks used to benchmark the SD scheduler.

B. Overhead Accounting

Runtime overheads can be accounted for in schedulability tests that assume overhead-free execution (*i.e.*, that consider overheads to be negligible) by inflating task parameters so that a positive overhead-free schedulability test of the transformed task set implies the timeliness of the original task set in the presence of overheads (*e.g.*, context switch costs can be charged to the preempting job as execution costs, *etc.*). The required transformations are conceptually simple, but somewhat tedious and beyond the scope of this paper; the interested reader may find a detailed description of the standard techniques applicable to GSN-EDF, GSN-EDF-DI, and SD in [13, Ch. 3].

However, G-EDF-MP requires additional accounting for the communication that it introduces. For the sake of completeness, we briefly document how we extended the accounting methods provided in [13] to reflect the added delays. We assume familiarity with the basic approach on behalf of the reader.

From an overhead accounting perspective, G-EDF-MP works similarly to GSN-EDF-DI. Thus, the accounting methods for scheduling (Δ^{sch}, Δ^{ipi}) and interrupt overheads (Δ^{rel}, Δ^{ev}) do not differ. However, every time a preemption occurs, the preempted lower-priority job is able to resume execution only after a full round-trip communication with the DSP. That is, **(i)** the client becomes idle, **(ii)** signals the completion, **(iii)** the DSP links a new task to the client and sends back an IPI. This situation is illustrated in Fig. 5, where the client processor becomes idle while task T_1 is pending.

Compared to the overheads delaying a preempted job under GSN-EDF or GSN-EDF-DI, this introduces additional delays due to the client request latency (Δ^{req}), the DSP handler (Δ^{dsp}), an additional IPI delay (Δ^{ipi}), and an additional scheduler invocation (Δ^{sch}) and context switch (Δ^{cxs}). This results in the following rule for inflating parameters.

For a sporadic task T_i with period p_i and execution cost e_i, let e'_i and p'_i denote the transformed parameters. Then

$$p'_i = p_i - \Delta^{ev}$$

and

$$e'_i = \frac{e_i + 3 \cdot (\Delta^{sch} + \Delta^{cxs}) + \Delta^{cpd}}{1 - U_0^{tck}} + 2 \cdot C^{pre} + \Delta^{req} + \Delta^{dsp} + 2 \cdot \Delta^{ipi} + \Delta^{rel},$$

where $C^{pre} = \frac{\Delta^{tck} + \Delta^{ev} \cdot U^{tck}}{1 - U^{tck}}$ denotes the interrupt-related cost per preemption [13, Sec. 3.4.5], and where $U^{tck} = \frac{\Delta^{tck}}{1000\mu s}$ denotes the utilization of the tick interrupt.