Hard Real-time Scheduling for Parallel Run-time Systems

Peter Dinda Xiaoyang Wang Jinghang Wang Chris Beauchene Conor Hetland
Northwestern University

ABSTRACT

High performance parallel computing demands careful synchronization, timing, performance isolation and control, as well as the avoidance of OS and other types of noise. The employment of soft real-time systems toward these ends has already shown considerable promise, particularly for distributed memory machines. As processor core counts grow rapidly, a natural question is whether similar promise extends to the node. To address this question, we present the design, implementation, and performance evaluation of a *hard* real-time scheduler specifically for high performance parallel computing on *shared memory* nodes built on x64 processors, such as the Xeon Phi. Our scheduler is embedded in a kernel framework that is already specialized for high performance parallel run-times and applications, and that meets the basic requirements needed for a real-time OS (RTOS). The scheduler adds hard real-time threads both in their classic, individual form, and in a group form in which a group of parallel threads execute in near lock-step using only scalable, per-hardware-thread scheduling. On a current generation Intel Xeon Phi, the scheduler is able to handle timing constraints down to resolution of ∼13,000 cycles (∼10 μs), with synchronization to within ∼4,000 cycles (∼3 μs) among 255 parallel threads. The scheduler isolates a parallel group and is able to provide resource throttling with commensurate application performance. We also show that in some cases such fine-grain control over time allows us to eliminate barrier synchronization, leading to performance gains, particularly for fine-grain BSP workloads.

CCS CONCEPTS

• **Software and its engineering → Real-time systems software**; **Runtime environments**; *Ultra-large-scale systems*;

KEYWORDS

hard real-time systems, parallel computing, HPC

ACM Reference Format:
Peter Dinda Xiaoyang Wang Jinghang Wang Chris Beauchene Conor Hetland. 2018. Hard Real-time Scheduling for Parallel Run-time Systems. In *HPDC '18: International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3208040.3208052

This project is made possible by support from the United States National Science Foundation through grant CCF-1533560 and from Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the United States Department of Energy's Office of Science.

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208052

1 INTRODUCTION

The performance of parallel systems, particularly at scale, is often highly dependent on timing and synchronization. Although timing and synchronization are very different concepts, we argue that for some parallel execution models, perfectly predictable timing behavior can substitute for synchronization. Perfectly predictable timing behavior can also be the cornerstone for achieving performance isolation within a time-sharing model, with its promise for better resource utilization. Finally, perfectly predictable timing behavior can be the basis for providing administrative control over resource utilization while maintaining commensurate application performance.

To achieve perfectly predictable timing behavior, we consider a fusion of parallel systems and hard real-time systems. Simply put, we investigate architecting a parallel application, its run-time system, and its OS as a hard real-time system. In previous work based on a *soft* real-time model (Section 7), we showed that a real-time model can allow time-sharing of single-node programs and virtual machines with controlled interference. We also showed it is possible to schedule a parallel application across a distributed memory parallel machine using centrally coordinated, but independent per-node real-time schedulers. In contrast, the present work focuses on a *hard* real-time model, applied with *no-holds-bared changes throughout the software stack* in a single node environment. Another important distinction is that we also consider extremely fine-grain scheduling, suitable, for example, for replacing some application-level barriers with timing.

We build upon prior work in hybrid run-time systems (HRTs), which have the premise of fusing a specialized kernel framework with a parallel run-time and its application to produce a custom kernel. We focus here solely on the integration of a hard real-time scheduling model into such an environment and its pluses and minuses. The specific environment we build within is the Nautilus kernel framework for building HRTs for x64. Our contributions are:

- We make the case for the fusion of parallel systems and hard real-time systems.
- We describe how the HRT environment and the Nautilus framework lay the groundwork for a hard real-time environment, as well as the limitations placed on such an environment by the hardware.
- We describe the design, implementation, and detailed performance evaluation of a hard real-time scheduler for node-level parallel systems. Our scheduler supports running groups of threads with fine-grain timing control and extremely close time-synchronization across CPUs.
- We describe techniques for ameliorating the central difficulties achieving individual and group hard real-time behavior on x64 platforms, namely system management interrupts

(SMIs) and other "missing time", as well as achieving coordinated measures of wall clock time (and thus time-based schedule synchronization) across CPUs.

- We evaluate the performance of the scheduler for a fine-grain bulk-synchronous parallel (BSP) benchmark. The scheduler is able to provide administrator resource control with commensurate application-level performance. Additionally, the scheduler's time-synchronization across CPUs make it possible to remove application barriers, leading to enhanced performance compared to a non-hard-real-time environment, especially as granularity shrinks.

The scheduler is implemented for all x64 machines, with our evaluation focusing primarily on the current generation Intel Xeon Phi. It is publicly available in the Nautilus codebase.

2 HYBRID RUN-TIMES (HRTS)

A hybrid run-time (HRT) is a mashup of an extremely lightweight OS kernel framework and a parallel run-time system [13, 16]. The desirability of this scheme is grounded in three observations. First, modern parallel run-time systems replicate a great deal of the functionality that a kernel typically provides. Second, these run-time systems are currently at the mercy of the abstractions and performance/isolation properties that the kernel, typically a commodity kernel, provides. The final observation is, by virtue of the run-time executing in user mode, the run-time cannot leverage the full hardware feature set of the machine; privileged features that it may be able to leverage are hidden behind a system call interface. In an HRT, all is kernel: the parallel run-time executes as a kernel in privileged mode, with full access to all hardware capabilities, and with full control over higher-level abstractions. The entire software stack, from the application down, executes with full privilege on native hardware.

Nautilus is a kernel framework designed to support HRT construction and research [14]. It is a publicly available open-source codebase that currently runs directly on x64 NUMA hardware, most importantly for this paper on the Knights Landing generation (the current generation) of the Intel Xeon Phi. Nautilus can help a parallel run-time ported to an HRT achieve very high performance by providing streamlined kernel primitives such as synchronization and threading facilities. It provides the minimal set of features needed to support a *tailored* parallel run-time environment, avoiding features of general purpose kernels that inhibit scalability.

At this point, ports of Legion, NESL, NDPC, UPC (partial), OpenMP (partial), and Racket have run in HRT form, fused with Nautilus. Application benchmark speedups from 20–40% over user-level execution on Linux have been demonstrated, while benchmarks show that primitives such as thread management and event signaling are orders of magnitude faster.

Nautilus can form the basis of a hard real-time system because its execution paths are themselves highly predictable. Examples that are particularly salient for the purposes of this paper include the following: Identity-mapped paging with the largest possible size pages are used. All addresses are mapped at boot, and there is no swapping or page movement of any kind. As a consequence, TLB misses are extremely rare, and, indeed, if the TLB entries can cover the physical address space of the machine, do not occur at all after startup. There are no page faults. All memory management, including for NUMA, is explicit and allocations are done with buddy system allocators that are selected based on the target zone. For threads that are bound to specific CPUs, essential thread (e.g., context, stack) and scheduler state is guaranteed to always be in the most desirable zone. The core set of I/O drivers developed for Nautilus have interrupt handler logic with deterministic path length. As a starting point, there are no DPCs, softIRQs, etc, to reason about: only interrupt handlers and threads. Finally, interrupts are fully steerable, and thus can largely be avoided on most hardware threads.

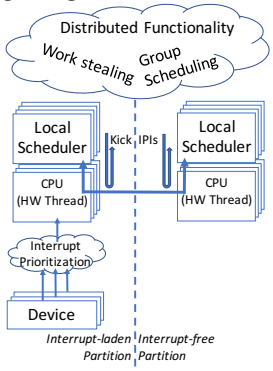


Figure 1: The global scheduler is a very loosely coupled collection of local schedulers, one per hardware thread/CPU. Most CPUs are are "interrupt-free" and see only scheduling-related interrupts. All CPUs are subject to SMIs at all times.

3 HARD REAL-TIME SCHEDULER

We have designed and implemented a new scheduler from the ground up for the Nautilus kernel framework. The scheduler's design and its integration with other components of the kernel framework are centered around the goal of achieving classic hard real-time constraints for the software threads mapped to an individual hardware thread (e.g., a hyperthread on x64 processors). While maintaining these constraints, the scheduler also implements non-real-time behavior such as work-stealing from other hardware threads, thread pool maintenance, and lightweight tasks.

We use the following terminology. A *CPU* is our shorthand way of referring to an individual hardware thread (hyperthread). A *local scheduler* is the scheduler of a CPU. The *global scheduler* is the distributed system comprising the local schedulers and their interactions. Figure 1 illustrates the global scheduler for the purposes of this section, while Figure 2 does the same for a local scheduler.

3.1 Model

Our scheduler as a whole adopts the classic model of Liu [25] for describing its interface, although not for its implementation.

An important concept is that of *admission control*. Before a thread is ever executed, it must indicate the timing constraints it requires to the scheduler. If the scheduler accepts these constraints, it guarantees that they will be met until the thread decides to change them, at which point the thread must repeat the admission control process. We refer to the wall clock time at which a thread is admitted as α, the admission time.

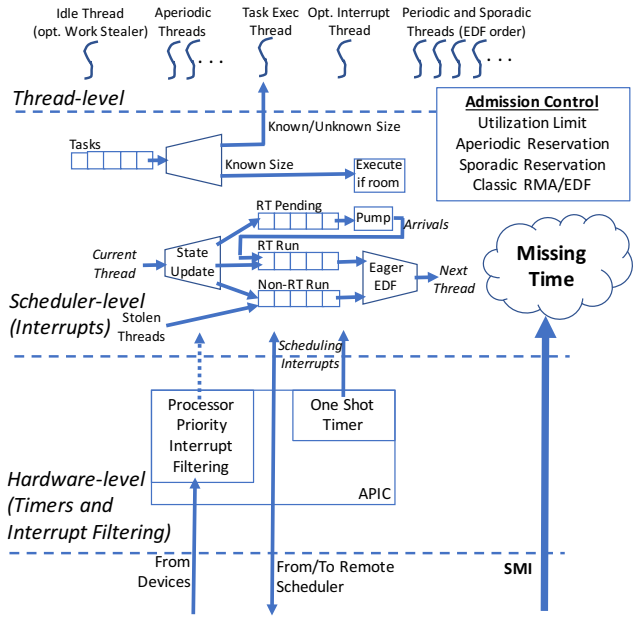


Figure 2: The local scheduler is a tightly coupled eager earliest deadline first (EDF) engine with additional support for non-real-time threads and tasks. Classic admission control approaches are used, with reservations and utilization limits added to account for the "missing time" of SMIs and scheduler overhead.

The following categories of timing constraints are observed:

- *Aperiodic* threads have no real-time constraints. They simply have a priority, μ. Newly created threads begin their life in this class.
- *Periodic* threads have the constraint (ϕ, τ, σ), which we refer to as phase (ϕ), period (τ), and slice (σ). Such a thread is eligible to execute (it *arrives*) for the first time at wall clock time $\alpha + \phi$, then arrives again at $\alpha + \phi + \tau$, $\alpha + \phi + 2\tau$, and so on. The time of the next arrival is the deadline for the current arrival. On each arrival the task is guaranteed that it will execute for at least σ seconds before its next arrival (its current deadline).
- *Sporadic* threads have the constraint $(\phi, \omega, \delta, \mu)$ or phase (ϕ), size (ω), deadline (δ), and aperiodic priority (μ). A sporadic thread arrives at wall clock time $\alpha + \phi$, and is guaranteed to execute for at least ω seconds before the wall clock time δ (the deadline), and subsequently will execute as an aperiodic thread with priority μ.

The scheduler also integrates support for finer granularity tasks, which have an even lower cost of creation, launching, and exiting than Nautilus threads. These are essentially queued callbacks implemented similarly to Linux's softIRQs or Windows DPCs, with an important exception in their processing. A task may be tagged with its size (ω). Size-tagged tasks can be executed directly by the scheduler (like softIRQs/DPCs) until a periodic or sporadic task arrives, but those without a size tag must be processed by a helper thread. As a consequence, the timing constraints of periodic and sporadic threads (e.g., the real-time threads) are not affected by tasks, and indeed, such threads are not even delayed by tasks.

3.2 Admission control

Admission control in our system is done by the local scheduler, and thus can occur simultaneously on each CPU in the system.

Aperiodic threads are always admitted. The particular meaning of their priority μ depends on which non-real-time scheduling model is selected when Nautilus is compiled.

Periodic and sporadic threads are admitted based on the classic single CPU schemes for rate monotonic (RM) and earliest deadline first (EDF) models [23]. Admission control runs in the context of the thread requesting admission. As a consequence, the cost of admission control need not be separately accounted for in its effects on the already admitted threads. This potentially allows more sophisticated admission control algorithms that can achieve higher utilization. We developed one prototype that did admission for a periodic thread-only model by simulating the local scheduler for a hyperperiod, for example.

At boot time each local scheduler is configured with a utilization limit as well as reservations for sporadic and aperiodic threads, all expressed as percentages. The utilization limit leaves time for the invocation of the local scheduler core itself in response to an timer interrupt or a request from the current thread. It can also be used for other interrupts and SMIs, as we describe below, but this is usually not necessary. The sporadic reservation provides time for handling spontaneously arriving sporadic threads, while the aperiodic reservation leaves time for non-real-time threads and for admission control processing.

3.3 Local scheduler and time

A local scheduler is invoked only on a timer interrupt, a kick interrupt from a different local scheduler, or by a small set of actions the current thread can take, such as sleeping, waiting, exiting, and changing constraints.

A local scheduler is, at its base, a simple earliest deadline first (EDF) engine consisting of a pending queue, a real-time run queue, and a non-real-time run queue. On entry, all newly arrived threads are pumped from the pending queue into the real-time run queue. Next, the state of the current thread is evaluated against the most imminent periodic or sporadic thread in the real-time run queue. If there is no thread on the real-time run queue, the highest priority aperiodic thread in the non-real-time run queue is used. A context switch immediately occurs if the selected thread is more important than the current thread.

The maximum number of threads in the whole system is determined at compile time, each local scheduler uses fixed size priority queues to implement the pending and real-time run queues, and other state is also of fixed size. As a result, the time spent in a local scheduler invocation is bounded. In other words, we can treat the local scheduler invocation itself as having a fixed cost. Bounds are also placed on the granularity and minimum size of the timing constraints that threads can request, limiting the possible scheduler invocation rate. Combining these limits allows us to account for scheduler overhead in the utilization limit selected at boot time.

Unlike typical EDF schedulers, our local scheduler is eager: if there is a real-time task that is runnable, we run it. We explain why in Section 3.6.

All time is wall clock time relative to the coordinated startup time of the local schedulers on all the CPUs. Measurement of time is done using the x64 cycle counter. As we describe in Section 3.4, cycle counters are synchronized at startup to the extent possible. Time is measured throughout in units of nanoseconds stored in 64 bit integers. This allows us at least three digit precision in our time computations and no overflows on a 2 GHz machine for a duration exceeding its lifetime.

When exiting an invocation, the local scheduler uses its CPU's advanced programmable interrupt controller's (APIC's) one-shot timer to select a time to be invoked again (like a "tickless" Linux kernel). At boot time, the APIC timer resolution, the cycle counter resolution, and the desired nanosecond granularity are calibrated so that the actual countdown programmed into the APIC timer will be conservative (resolution mismatch results in earlier invocation, never later). If the APIC supports "TSC deadline mode" (some Intel processors), it can be programmed with a cycle count instead of an APIC tick count, avoiding issues of resolution conversion.

3.4 Global scheduler and interactions

A local scheduler is almost entirely self contained, which makes possible the use of simple, classic admission control and scheduling. Each local scheduler is also driven independently by a separate timer as each CPU has its own APIC. To the greatest extent possible, all local schedulers coordinate using wall clock time.

At boot time, the local schedulers interact via a barrier-like mechanism to estimate the phase of each CPU's cycle counter relative to the first CPU's cycle counter, which is defined as being synchronized to wall clock time. Since the kernel starts its boot process on each CPU at a slightly different time, and may require different amounts of time to boot on each one, this calibration is vital to having a shared model of wall clock time across the system. In machines that support it, we write the cycle counter with predicted values to account for the phase difference, attempting to bring them as close to identical as possible.

As both the phase measurement and cycle counter updates happen using instruction sequences whose own granularity is larger than a cycle, the calibration does necessarily have an error, which we then estimate and account for when a local scheduler uses its own cycle counter as an estimate for the wall clock time.

We require that the processor provide constant cycle counter behavior ("constant TSC"—a common feature), and that the execution of below-kernel firmware does not stop or manipulate the cycle counter. We do not currently support dynamic voltage and frequency scaling or above-maximum clock rates (e.g., "turboboost") although the design does have room to handle it in the future.

Figure 3 shows a histogram of the typical cycle counter offsets our calibration logic derives on the Xeon Phi KNL using our techniques. We estimate that we bring all 256 CPUs to an agreement about wall clock time that is accurate to about 1000 cycles (slightly less than a microsecond on this hardware).

Each local scheduler has lockable state, and a scheduler invocation takes the lock, and thus could conceivably wait on some other lock holder. However, we have designed this to only be possible during a small set of activities such as thread reaping/reanimation (thread pool management), work stealing, and garbage collection.

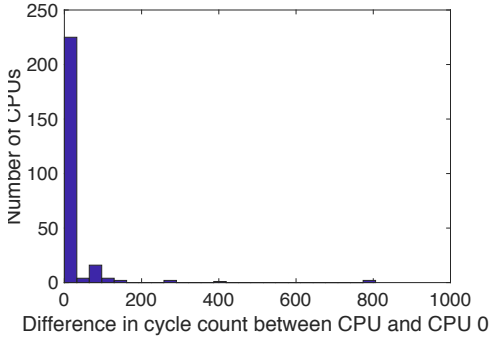


Figure 3: Cross-CPU cycle counter synchronization on Phi. We keep cycle counters within 1000 cycles across 256 CPUs.

Each of these features, with the exception of garbage collection, in turn holds a local scheduler's lock for a time bounded by the compile-time selected limit on the total number of threads. Furthermore, each of these features can be disabled at compile-time, allowing us to configure a kernel where local scheduler locking occurs only within an invocation of the local scheduler.

The work stealer, which operates as part of the idle thread that each CPU runs, uses power-of-two-random-choices victim selection [28] to avoid global coordination, and only locks the victim's local scheduler if it has ascertained it has available work. Only aperiodic threads can be stolen or otherwise moved between local schedulers. This avoids the need for any parallel or distributed admission control, and simplifies the implementation of hard real-time group groups (Section 4).

3.5 Interrupt steering and segregation

In order to achieve hard real-time constraints, a scheduler must either control or be able to predict all time-consuming activities in the system. Interrupts, in which we include exceptions, traps, aborts, system calls, interprocessor interrupts (IPIs) and external interrupts, must therefore be accounted for. Nautilus has no system calls as all code executes in kernel mode. Exceptions, traps, and aborts are treated as hard, fail-stop errors, and thus can be ignored for scheduling purposes. This leaves IPIs and external interrupts.

Due to the design of Nautilus, IPIs are rarely needed. For example, as there is a single system-wide address space that is configured at boot time and then never changes, no TLB shootdowns are ever done. In the configuration used in this paper, IPIs are only used to communicate between local schedulers, to allow one local scheduler to "kick" another to get it to do a scheduling pass. The cost of a scheduling pass is bounded, as we described previously, as is the kick rate. As a consequence, the cost of IPIs is known and is taken into account via the limit on utilization.

An external interrupt, from an I/O device, for example, can be steered to any CPU in the system. Using this mechanism we can partition the CPUs into those that receive external interrupts (the interrupt-laden partition) and those that do not (the interrupt-free partition). External interrupts are then a non-issue for scheduling in the interrupt-free partition. The default configuration is that the interrupt-laden partition consists of the first CPU, but this is can be changed according to how interrupt rich the workload is.

Even for the interrupt-laden partition, we ameliorate interrupts through two mechanisms. First, the local scheduler sets the CPU's APIC's hardware processor priority as part of switching to a new thread. For a hard real-time thread, the priority is set such that the only interrupts that the thread will see are scheduling related (i.e., the APIC timer, and the "kick" IPI). These are already accounted for in scheduling admission control decisions. In other words, this mechanism steers interrupts away from the hard real-time threads. In contrast, the second mechanism provides the ability to steer interrupts toward a specific "interrupt thread". Note that for both of these mechanisms, the allowed starting time of an interrupt is controlled, however the ending time is not. Thus to use these mechanisms to preserve hard real-time behavior on the interrupt-laden partition, interrupt handler time and invocation rates must be bounded, known, and accounted for in setting the utilization limit of the local scheduler. The device drivers we have built so far for Nautilus have thus been designed with the principle of having a bounded interrupt handling time.

Another mechanism we use to avoid having interrupts affect hard real-time operation of threads is eager scheduling, which we describe within the next section.

3.6 SMIs and mitigation of "missing time"

A major impediment to achieving hard real-time behavior on x64 machines is the existence of system management mode (SMM) and interrupts (SMIs). SMM, which has been a part of Intel processors since the early 1990s, boils down to the ability of the system firmware (BIOS) to install a special interrupt handler (the SMI handler) that vectors into firmware. This handler cannot then be changed, or masked, by any higher-level software such as a VMM or kernel. The handler's code and state are curtained by the memory controller under normal operation. When the system hardware generates an SMI, all CPUs stop, the memory controller uncurtains the SMM memory, and one CPU then executes the SMI handler. Once it finishes, it executes a special return-from-interrupt instruction that again curtains the memory, and resumes all of the CPUs.

Manufacturers of systems (and processors) use SMM code and SMIs for various purposes and their incidence and cost vary across systems. From the perspective of even the lowest level kernel or VMM code, SMIs appear to be "missing time"—at unpredictable points, the cycle counter appears to advance by a surprisingly large amount, meaning that a significant amount of wall clock time has passed without software awareness.[1] From the parallel systems perspective, SMIs are "OS noise" [10] that the OS simply cannot control. The missing time of SMIs can have dramatic effects on overall system behavior even in general purpose systems [5].

For a hard real-time scheduler, the missing time of SMIs can be catastrophic. If SMI rates and handling times are unpredictable or unbounded, the scheduler *cannot* guarantee *any* hard real-time constraint will be met. However, for any real hardware, these cannot be unbounded as this would imply the system could not make forward progress or that the user experience could be negatively impacted by glitches. The issue is estimating the bounds. Note that having unpredictable or unbounded handling times and rates for ordinary interrupts presents the same problem as SMIs.

We address this problem through eager, work-conserving scheduling. In many hard real-time schedulers, a context switch to a newly arrived thread is delayed until the last possible moment at which its deadline can still be met. The goal is in part to create opportunities to handle newly arriving threads that might have an even more imminent deadline. While this non-work-conserving behavior is ideal, the consequence of missing time due to SMIs (or difficult to predict ordinary external interrupt costs in the interrupt-laden partition) is that the thread may be resumed at a point close to its deadline, but then be interrupted by an SMI that pushes the thread's completion past its deadline.

In our local scheduler, in contrast, we never delay switching to a thread. Consequently, while missing time introduced by SMIs or badly predicted interrupts can still push back the thread's completion time, the probability of pushing it back beyond the deadline is minimized—We start early with the goal of ending early even if missing time occurs. The utilization limit then acts as a knob, letting us trade off between between sensitivity to SMIs/badly predicted interrupts, and utilization of the CPU.

4 HARD REAL-TIME GROUPS

As described to this point, our scheduler supports individual hard real-time (and non-real-time) threads. Parallel execution demands the ability to collectively schedule a group of threads running across a number of CPUs. That is, we want to gang schedule the group of threads. In our system we achieve this through a distributed, time-based mechanism that builds on local hard real-time behavior. Our mechanism has the benefit of not requiring any communication between the local schedulers once all the threads in the group have been admitted. All coordination costs are borne at admission time.

4.1 Scheduler coordination

To understand why it is possible to effectively gang-schedule a group of threads with no communication among the local schedulers, consider the following. Let us have a pair of local schedulers, A and B, that have the same configuration (utilization limits, interrupt steering, reservations for sporadic and aperiodic threads, etc.) Consider further that both A and B are currently each handling a set of threads, S_A and S_B, where the timing constraints of the threads in S_A are the same as those in S_B. That is, while A and B are scheduling different sets of threads, they are attempting to meet exactly the same set of constraints. Because our local scheduler is completely deterministic by design, both A and B are making identical scheduling decisions on each scheduling event (e.g., timer interrupt). Hence, in a particular context in which A switches to a particular thread, B would also switch to the parallel thread (the one with identical constraints to A's) in its set.

Given that the constraints of S_A and S_B are the same, it is not only the case that A and B are making the same decisions, but also that they are making them *at the same time*; all local schedulers in the system are driven by wall clock time.

Now consider adding a new pair of threads, a and b, to the respective local schedulers. If a and b have identical constraints, then both local schedulers are given the same admission control

[1] Much as in an alien abduction in classic UFOlogy, except that the alien in question is the hardware vendor, and the abductee is the kernel.

```
conduct leader election;
if I am the leader then
    lock group;
    attach constraints to group;
end
execute group barrier;
conduct local admission control;
execute group reduction over errors;
if any local admission control failed then
    readmit myself using default constraints;
    execute group barrier;
    if I am the leader then
        unlock group;
    end
    return failure;
end
execute group barrier and get my release order;
phase correct my schedule based on my release order;
if I am the leader then
    unlock group;
end
return success;
```

Algorithm 1: Group admission control algorithm.

request with the same current set of constraints. It must be the case that if the new constraints are realizable, then both local schedulers will admit the threads, and, from that point in time, make identical decisions about them. Several subtleties do arise here, which we describe how to mitigate in Section 4.4.

4.2 Groups

We have added a thread group programming interface to Nautilus for group admission control and other purposes. Threads can create, join, leave, and destroy named groups. A group can also have state associated with it, for example the timing constraints that all members of a group wish to share. Group admission control also builds on other basic group features, namely distributed election, barrier, reduction, and broadcast, all scoped to the group.

4.3 Group admission control

Group admission control is done by having every thread in a group call a single function which parallels the function called for individual admission control. Instead of invoking

```
nk_sched_thread_change_constraints(constraints);
```

to change the caller's constraints, each member of the group of threads instead invokes

```
nk_group_sched_change_constraints(group, constraints);
```

This function either succeeds or fails for all the threads.

The pseudocode for the function is straightforward and is shown in Algorithm 1. The "default constraints" refer to aperiodic constraints. Admission control for aperiodic threads cannot fail in our system, so aperiodic constraints are used as fallbacks.

As can be readily seen, the algorithm is quite simple. The heavy lifting is accomplished by the local admission control process. Also note that coordination is limited to the barriers and reductions, and these are essentially the limiters on scalability of the group admission control process, as we will see. Note again that, once admitted, no further coordination is done, except via time.

4.4 Phase correction

Several issues arise in our group scheduling scheme, all of which we ameliorate through careful control of the phase parameter (ϕ) for individual periodic or sporadic threads (Section 3.1).

The first issue is that during the local admission control process for an individual thread, the admission control algorithm runs in the context of the thread, and the thread is aperiodic (not real-time), and hence could be delayed. Furthermore, in an interrupt-laden partition an external interrupt may also delay the admission control processing of the local scheduler. Even in an interrupt-free partition, SMIs can still cause such delays. As a consequence, the wall-clock time at which the admission processing finishes (α) may be slightly different on two local schedulers given identical threads being admitted at the same time.

Barriers among the threads are used in the group admission control process to help ameliorate this issue, but they introduce their own problem: threads are not released from a barrier at identical times. There is some first thread to leave the barrier, and some last thread. The gap in time between them effectively becomes a gap in the actual wall clock time their admission control processing completes, even if their αs are identical.

The final issue is that wall clock time is not (and cannot be) identical between local schedulers. Time cannot be exactly synchronized in a distributed system. Figure 3 suggests in our system, on a Phi, it can be synchronized to less than 1000 cycles, but this is not zero. As a consequence, the actual wall clock time represented by α on one CPU may occur slightly behind the actual wall clock time represented by the same α on another.

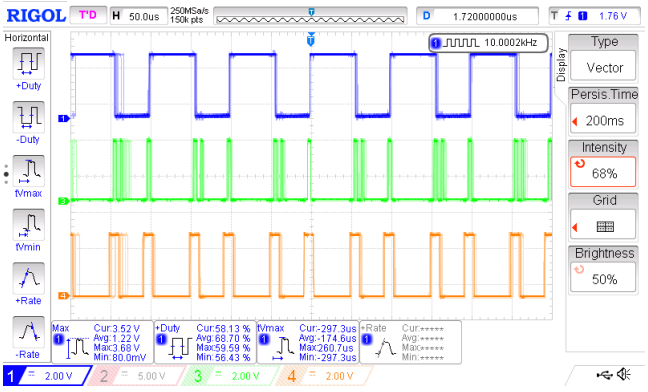
To further mitigate both of these problems, we use the thread phase, ϕ, to get thread scheduling events across the CPUs to be synchronized to the limits possible given the synchronization of the wall clock time. Recall that a sporadic or periodic thread first arrives at wall clock time $\alpha + \phi$. For each thread in a group, we detect its release order i for its final barrier before it becomes a real-time thread. The ith thread to be released is then given a corrected phase $\phi_i^{corrected} = \phi + (n - i)\Delta$ where Δ is the measured per-thread delay in departing the barrier.

5 EVALUATION

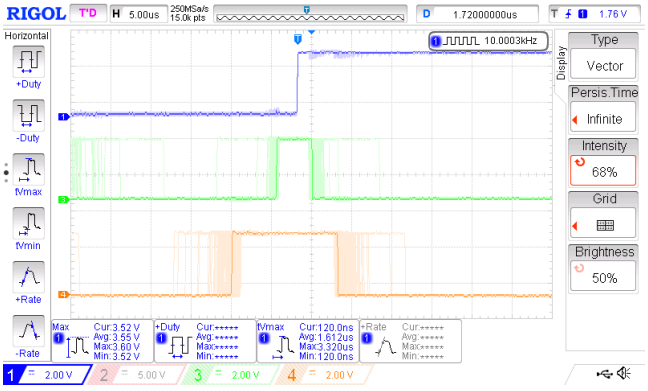
We evaluated the scheduler for correctness and for the limits of its performance for scheduling individual threads and for admitting, scheduling, and keeping groups of threads synchronized.

5.1 Testbed

Unless otherwise noted, our evaluation was carried out on a Colfax KNL Ninja platform. This is an Intel-recommended platform for Xeon Phi Knights Landing (KNL) development. It is essentially a Supermicro 5038ki, and includes a Intel Xeon Phi 7210 processor running at 1.3 GHz. The processor has 64 cores, each of which has 4 hardware threads (CPUs in our terminology). The processor is tightly coupled to 16 GB of MCDRAM, and more loosely to 96 GB of conventional DRAM. All visible CPU throttling/burst behavior is disabled in the BIOS.



(a) Broad view



(b) Single interrupt

Figure 4: Hard real-time scheduling in Nautilus on Phi as measured by external scope. From top to bottom, the traces are the test thread, the local scheduler pass, and the interrupt handler (which includes the scheduler pass and context switch).

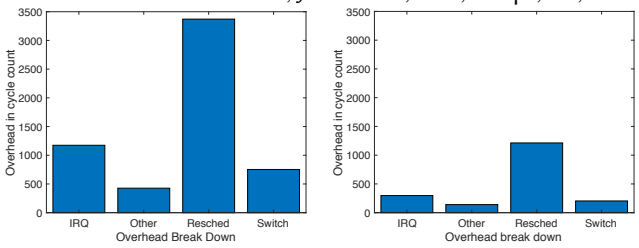
We conducted an identical evaluation on Dell R415 hardware consisting of dual AMD 4122 processors running at 2.2 GHz (8 hardware threads / cores total) and 16 GB DRAM. Since the individual hardware threads on this machine are much faster than individual Phi hardware threads, even finer grain scheduling and synchronization is possible than shown here for the Phi.

Unless otherwise noted, we configure the scheduler with a default configuration (99% utilization limit, 10% sporadic reservation, 10% aperiodic reservation). We schedule non-real-time aperiodic threads using round-robin scheduling with a 10 Hz timer, which given the evaluation is the best configuration outside of non-preemptive scheduling.

5.2 External verification

A hard real-time scheduler, because it operates in sync with wall clock time, must be verified by timing methods external to the machine. To do so, we added a parallel port interface (a GPIO interface) to our machine. A single `outb` instruction within a local scheduler is sufficient to change all 8 pins on the output connector. We monitor those pins using an oscilloscope (a Rigol DS1054Z, which is a 4 channel, 50 MHz, 1 GS/s DSO).

Figure 4 shows scope output when a local scheduler is running a periodic thread with period $\tau = 100\mu s$ and slice $\sigma = 50\mu s$. Figure 4(a)



(a) Phi (b) R415

Figure 5: Breakdown of local scheduler overheads on Phi and R415.

and (b) are showing the same experiment. In (a), we are looking at a 600 μs window with trace persistence at 200 ms. In (b), we zoom in on this, looking at a 60 μs window around a timer interrupt, and with trace persistence set to infinity. Recall that an oscilloscope draws the trace at full speed again and again. The trace fades according to the persistence value. Any fuzziness in the trace thus corresponds to deviation from "normal" behavior. The three traces are, from top to bottom, the active and inactive times of the test thread, the active and inactive times of the scheduler, and the active and inactive times of the timer interrupt. The timer interrupt times include the interrupt overhead, the scheduler pass, and a context switch.

The test thread is marked as active/inactive at the end of the scheduler pass, hence its active time includes the scheduler time, which is why the duty cycle is slightly higher that 50%. Also, on every second cycle, the scheduler runs twice, once due to the interrupt, and once again shortly thereafter. This is because when we switch from the test thread to the idle thread, the idle thread runs the work-stealer and then immediately yields to any new work. Since here there is no work to steal, the scheduler simply decides to continue running the idle thread.

The most important observation here is that while the interrupt handler trace (bottom) and scheduler trace (middle) have "fuzz", the scheduler keeps the test thread trace (top) sharp. This is particularly easy to see in Figure 4(b). This combination of fuzziness and sharpness tells us that the scheduling algorithm is doing its job of controlling time (other threads, interrupts, etc) so as to meet the test thread's timing constraints, and to meet them as deterministically as possible.

We evaluated various combinations of period and slice using this method and found the same, correct behavior. Provided the period and slice are possible once the time costs of the scheduler pass, timer interrupt handling, and thread context switch are accounted for, the timing constraints will always be met.

5.3 Local scheduler overheads and limits

Due to its distributed nature, our scheduler's overheads, and thus the limits on the possible timing constraints (i.e., on ϕ, τ, σ, δ and ω), are largely determined by the local scheduler's overheads. Figure 5(a) breaks down the local scheduler overhead on the Phi that is incurred on a timer interrupt. These times are measured using the cycle counter on the CPU, and do not account for interrupt dispatch overheads in the processor. They represent the overheads that could be changed through software means.

On the Phi, the software overhead is about 6000 cycles. For a periodic thread, two interrupts are incurred on each period (the

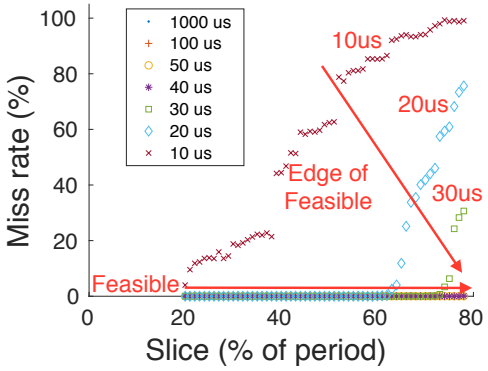


Figure 6: Local scheduler deadline miss rate on Phi as a function of period (τ) and slice (σ). Once the period and slice are feasible given scheduler overhead, the miss rate is zero.

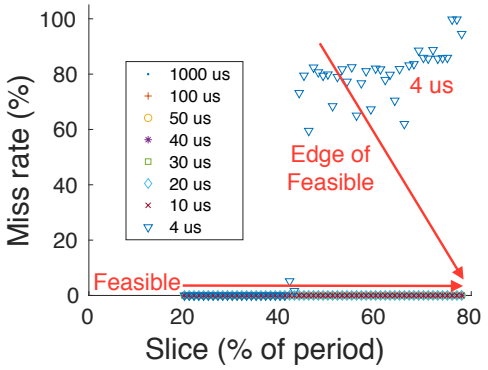


Figure 7: Local scheduler deadline miss rate on R415 as a function of period (τ) and slice (σ). Once the period and slice are feasible given scheduler overhead, the miss rate is zero.

arrival, and the timeout), although these can overlap (one thread's timeout and the next thread's arrival can be processed in the same interrupt.) Hence, the limits on scheduling constraints will be in the 6000-12000 cycle range (4.6 to 9.2 μs). About half of the overhead involves the scheduling pass itself, while the rest is spent in interrupt processing and the context switch.

Figure 6 shows the the local scheduler miss rate on the Phi. Here we have turned off admission control to allow admission of threads with infeasible timing constraints. Each curve represents a different period, the x-axis represents the slice as a fraction of the period, and the y-axis shows the miss rate. On this kind of graph, we expect a sharp disconnect: for too small of a period or slice, or too large of a slice within a period, misses will be virtually guaranteed since the scheduler overhead will not leave enough time. On the other hand, once the period and slice are feasible given the scheduler overhead, we expect a zero miss rate. As the graph shows, the transition point, or the "edge of feasibility" is for a period of about 10 μs, as we would expect given the overhead measurements.

It is important to realize that an individual Phi CPU is quite slow, both in terms of its clock rate and its superscaler limits. On faster individual CPUs, the scheduling overheads will be lower in terms of both cycles and real time, as can be seen for the R415 in Figure 5(b).

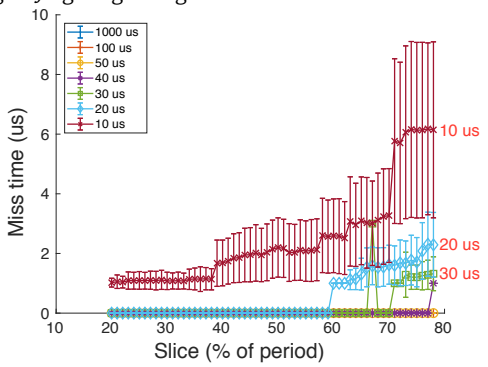


Figure 8: Average and standard deviation of miss times for feasible schedules on Phi. For infeasible constraints, which are usually not admitted, deadlines are missed by only small amounts.

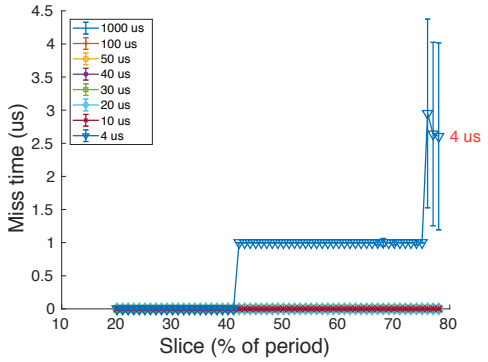


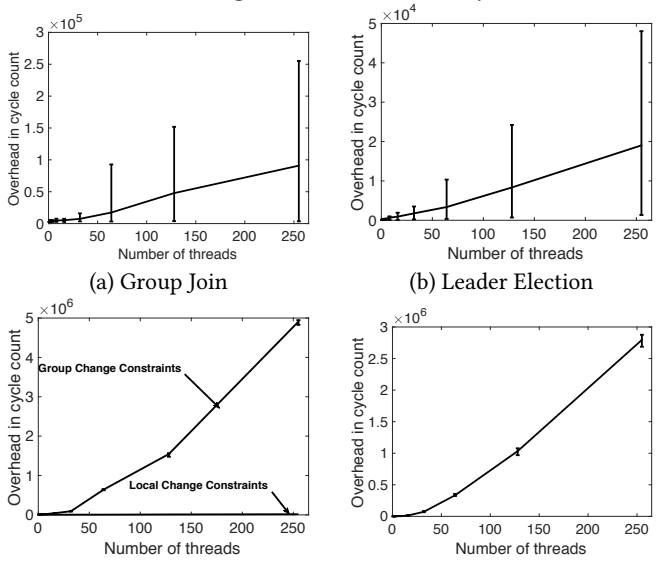
Figure 9: Average and standard deviation of miss times for feasible schedules on R415. For infeasible constraints, which are usually not admitted, deadlines are missed by only small amounts.

These lower overheads in turn make possible even smaller scheduling constraints, as can be seen for the R415 in Figure 7. Here, the edge of feasibility is about 4μs.

It is also instructive to see what happens beyond the edge of feasibility, when deadlines misses occur because the timing constraint is simply not feasible given the overhead of the scheduler. In Figures 8 (Phi) and 9 (R415) we show the average and variance of miss times both for feasible and infeasible timing constraints. For feasible timing constraints, the miss times are of course always zero. For infeasible timing constraints, the miss times are generally quite small compared to the constraint. Note again that, in normal operation, infeasible constraints are filtered out by admission control.

5.4 Group admission control costs

In steady state operation, once a group of threads has been admitted in our system, only the overheads of the local schedulers matter. These are quite low, as shown previously. The time costs of group operation are born solely when the group of threads is admitted, namely for the algorithm in Section 4.3. Figure 10 breaks down the absolute costs of the major steps of the algorithm: group join, leader election, distributed admission control, and the final barrier.



(a) Group Join (b) Leader Election

(c) Distributed Admission Control (d) Barrier/Phase correction

Figure 10: Absolute group admission control costs on Phi as a function of the number of threads in the group. The average time across the threads, as well as the minimum and maximum times encountered by individual threads is shown.

The average time per step grows linearly with the number of threads because we have opted to use simple schemes for coordination between local schedulers during group admission control. Even still, the absolute costs at up to 255 threads are minimal and dominated by the group admission control and final barrier steps. Only about 8 million cycles (about 6.2 ms) are needed at 255 threads. More sophisticated coordination schemes would likely make the growth logarithmic with the number of threads, should this be needed in a future node with many more CPUs.

Note that the core of the algorithm is local admission control. Figure 10(c) shows curves for both global admission control ("Group Change Constraints") and the local admission control ("Local Change Constraints") that it builds on. The local admission control cost is constant and independent of the number of threads. This is the hard limit to the cost of global admission control.

5.5 Group synchronization limits

Once a group of threads has been admitted, the local schedulers' individual operation, coordinated only via wall clock time, should suffice to keep the threads executing in near lock step. Figure 11 illustrates this in operation. Here, an 8 thread group, one thread per CPU, has been admitted with a periodic constraint. Each time a local scheduler in invoked and context-switches to a thread in the group, it records the time of this event. A point in the graph represents the maximum difference between the times of these events across the 8 local schedulers. For example, a point at (4000,7500) means that at scheduling event 4000, the slowest scheduler was operating 7500 cycles behind the fastest one.

In this and the subsequent graphs of this section, it is important to understand that phase correction is disabled, hence there is a bias. In this figure, the "first" member of the group is on average about 5000 cycles ahead. This average bias is eliminated via phase correction.

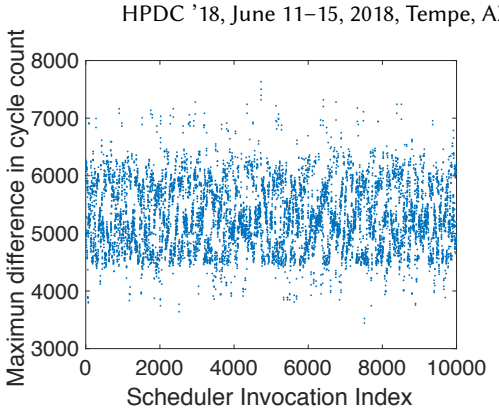


Figure 11: Cross-CPU scheduler synchronization in an 8 thread group admitted with a periodic constraint on Phi. Context switch events on the local schedulers happen within a few 1000s of cycles.

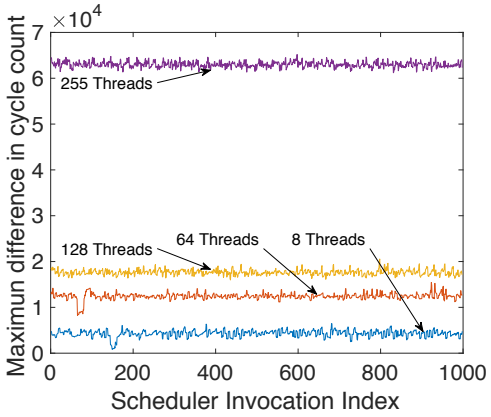


Figure 12: Cross-CPU scheduler synchronization in different size groups with periodic constraints. The average difference, which depends on the number of threads in the group, can be handled with phase correction. The more important, and uncorrectable, variation is on the other hand largely independent of the number of threads in the group.

What is important in the figure (and later figures) is the *variation*. Here, we can see that timing across the threads has a variation of no more than 4000 cycles (3 μs). This variation, which phase correction cannot correct, is largely independent of the number of threads. Recall that time synchronization itself has a variation of about 1000 cycles (Figure 3), hence the group scheduler is approaching the thread synchronization limits possible on the hardware.

Figure 12 extends these results, in a less busy graph, to groups with up to 255 threads. As before, it is the uncorrectable variation that matters, and this is largely independent of group size. Even in a fully occupied Phi, with one CPU in the interrupt-laden partition and 255 in the interrupt-free partition, we can keep threads in the latter partition synchronized to within about 4000 cycles (3 μs) purely through the use of hard real-time scheduling.

6 BSP EXAMPLE

To consider how the hard real-time scheduling model and our scheduler could benefit applications, we implemented a microbenchmark and then used it to consider two potential benefits of our model:

- Resource control with commensurate performance: Can we throttle up and down the CPU time resource given to a parallel application with proportionate effects on its performance?
- Barrier removal: Can we use real-time behavior to avoid barriers that are necessary in a non-real-time model?

We can answer yes to both questions.

6.1 Microbenchmark

We developed a bulk-synchronous parallel (BSP [11]) microbenchmark for shared memory that allows fine grain control over computation, communication, and synchronization. The benchmark emulates iterative computation on a discrete domain, modeled as a vector of doubles.

As used here, the algorithm is parameterized by `P`, the number of CPUs used (each CPU runs a single thread), `NE`, the number of elements of the domain (vector) that are local to a given CPU, `NC`, the number of computations done on each element per iteration, `NW`, the number of remote writes to do to other CPUs' elements per iteration, and `N`, the number of iterations done in total. In the following, remote writes are done according to a ring pattern. CPU `i` writes to some of the elements owned by CPU `(i+1) % P`.

Once the threads are running, they execute an `nk_group_sched_change_constraints()` call to use a common schedule, and to synchronize their operation. Next, they all execute the following:

```
for (i=0; i<N; i++) {
  for (j=0; j<NE; j++) {
    compute_local_element(NC);
  }
  optional_barrier();
  for (j=0;j<NW;j++) {
    write_remote_element_on((myproc+1)%P);
  }
  optional_barrier();
}
```

As is common to BSP codes, the barriers in the above are needed for any non-real-time schedule since we need to assure that each compute or communication phase completes before the next phase starts to avoid race conditions on the elements. With a group real-time schedule, this property may be feasible to provide via timing.

6.2 Study

We ran a parameter study using the microbenchmark and our scheduler on the Xeon Phi. We in particular wanted to study the part of the parameter space where the compute and communicate phases in the algorithm are short. These create the greatest stress for the scheduler for providing resource control with commensurate performance, and present the most interesting opportunities for barrier removal.

We swept `P` from 1 to 255 CPUs in powers of two, and `NE`, `NC` and `NW` from 1 to 128 in powers of two. `N` was chosen to be large enough in all cases to allow us to see the steady state behavior of the scheduler's interaction with the benchmark. All threads are mapped 1:1 with CPUs in the interrupt-free partition.

We considered both aperiodic and periodic group constraints. For aperiodic, since no other threads are available on the CPUs, the benchmark has as much CPU time as possible—there are no context switches and scheduling interrupts are extremely rare (they occur at 10 Hz, similar to Kitten [20]). We considered 900 different periodic constraints, with the period ranging from 100 μs to 10 ms in steps of 100 μs, and, for each period, 9 slices comprising 10, 20, ..., 90% of the period. A final parameter involves executing, or not executing, the `optional_barrier()` call.

The entire parameter space explored consists of 1,037,852 combinations. In the following, we focus on the cases where there are 255 CPUs (the largest scale), and where `NC` and `NW` are equal (indicating a roughly 50/50 compute/communicate ratio). The value for `NE` then essentially gives us the computation granularity. Our observations can be readily understood simply by considering the extremes of granularity ("coarsest" and "finest").

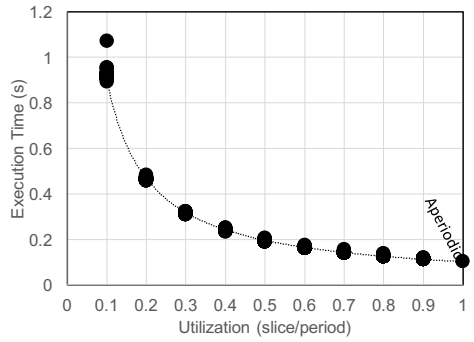


Figure 13: Resource control with commensurate performance for coarsest granularity with barriers. All period and slice combinations are plotted, with utilization meaning the ratio of slice to period (σ/τ). Regardless of the specific period chosen, benchmark execution rate matches the time resources given.

6.3 Resource control

The group hard real-time scheduling scheme can keep application threads executing in lock-step. As a consequence, when one thread needs to synchronize or communicate with another, for example via the `optional_barrier()` in our benchmark, the synchronization is almost certain to avoid any blocking on a descheduled thread. Further note that the scheduler is providing isolation. While this isolation is limited to timing, as we previously described [6, 22], timing isolation, combined with other readily available resource isolation techniques available at the OS level and above, can result in quite strong isolation properties across most resources.

These observations suggest that the period (τ) and slice (σ) constraints of a periodic real-time constraint can be used to control the resource utilization (utilization is $\frac{\sigma}{\tau}$) while providing commensurate application performance. That is, if $\frac{\sigma}{\tau} = 0.5$, we expect the application to operate at 50% of its top speed, not slower. Our prior work also showed how to make this possible in a distributed environment for relatively coarse granularities using soft real-time scheduling with feedback control. Does our hard real-time scheduling approach make this possible for fine granularities in a shared memory machine?

Figure 13 clearly shows that this is indeed the case. Here, we are looking at the coarsest granularity computation, with barriers,

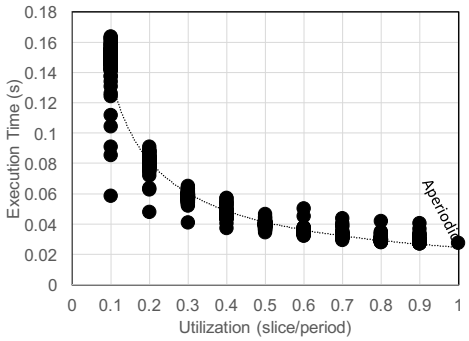


Figure 14: Resource control with commensurate performance for a finest granularity with barriers. All period/slice combinations are plotted, with utilization meaning the ratio of slice to period (σ/τ). Regardless of the specific period chosen, benchmark execution rate roughly matches the time resources given.

running on 255 CPUs. In the figure, each point represents a single combination of period and slice (there are 900), plus aperiodic scheduling with 100% utilization. Regardless of the period selected, the performance of the benchmark is cleanly controlled by the time resources allocated. As the granularity shrinks, proportionate control remains, as can be seen in Figure 14, in which we consider the finest granularity in our study. In this case, there is more variation across the different period/slice combinations with the same utilization because the overall task execution time becomes similar to the timing constraints themselves for some of the combinations.

6.4 Barrier removal

Because of the lock-step execution across CPUs that our scheduler can provide for hard real-time groups, and the fully balanced nature of a BSP computation as modeled in our microbenchmark, it is possible to consider discarding the optional barriers. What are the performance benefits of doing so?

The benefits depend on the granularity of the computation. Considering the BSP structure as represented in the benchmark, Amdahl's law tells us that the cost of `optional_barrier()` only matters if the costs of `compute_local_element()` and `write_remote_element_on()` are small. `optional_barrier()`'s cost is only dependent on the number of CPUs, so this boils down to how large `NE`, `NC`, and `NW` are.

Figure 15 shows us the benefit for the coarsest granularity computation, on 255 CPUs. In the figure, each point represents a single combination of period and slice (there are 900). The point compares the time with the barrier to the time without. All points above the line (almost all of them) represent configurations where the benchmark is running faster without the barrier. The points form 9 clusters as these are the 9 levels of slice (10%, 20% and so on) in our study. Also indicated is the performance of the aperiodic constraints (which must be run with barriers for correctness). With a 90% slice (utilization), the hard real-time scheduled benchmark, with barriers removed, matches and sometimes slightly exceeds the performance of the non-real-time scheduled benchmark. The latter is running at 100% utilization.

Figure 16, which is semantically identical, shows the benefit for the finest granularity computation on 255 CPUs Here, the benefit of

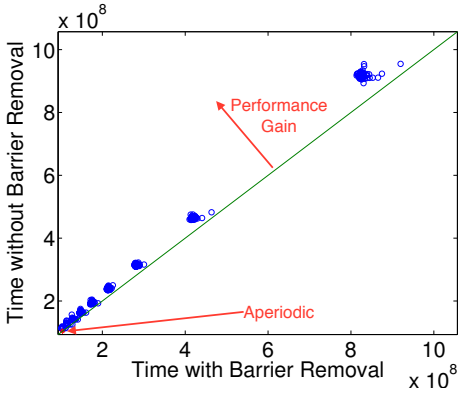


Figure 15: Benefit of barrier removal at the coarsest granularity. Distance of points above line indicates benefit. The real-time scheduled benchmark without barriers and with period/slice constraints giving 90% utilization, is similar in performance to the non-real-time scheduled benchmark with barriers at 100% utilization. Time in ns.

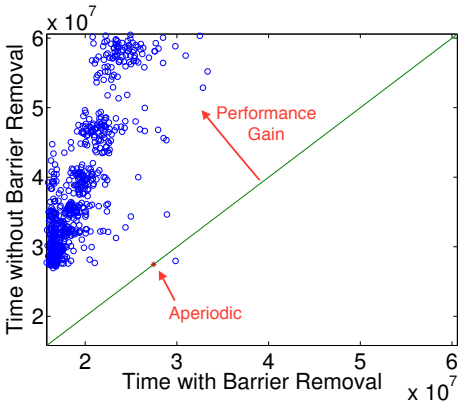


Figure 16: Benefit of barrier removal for the finest granularity. Distance of points above line indicates benefit. For a range of period/slice combinations and utilizations, the real-time scheduled benchmark without barriers exceeds the performance of the non-real-time scheduled benchmark with barriers. Time in ns.

barrier removal is much more pronounced, as Amdahl's law would suggest, and the effects of barrier removal are much more varied. The benefit ranges from about 20% to over 300%. As before the aperiodic/100% utilization case, which requires the barrier, is highlighted. Now, however, the hard real-time cases, with barriers removed, can not just match its performance, but in fact considerably exceed it.

7 RELATED WORK

Our work ties to long-running research in several areas, as described below. It is important to keep in mind that we target fine-grain scheduling of CPUs in a shared memory node, not courser grain scheduling of a distributed memory machine.

OS noise, overhead, and responses. In distributed memory parallel machines, it has been well known for a long time that operating system noise can produce differentiated timing, and ultimately performance degradation [10] and variation [19]. These issues compound as scale increases, as do the overheads of using full OS

kernels on such machines. One response has been specialized operating systems, including the long-term development of lightweight kernels from the early 90s [1, 40] through today [20, 36]. Lightweight kernels can provide both low overheads and a high degree of predictability for parallel applications.

The hybrid run-time (HRT) model [16] and Nautilus kernel framework [14] that we build upon in this paper extend the lightweight kernel concept by merging the application, parallel run-time, and kernel into a single entity (the HRT). Such an approach is becoming increasingly feasible through the combination of a rapidly growing core counts and tools that make it possible to boot (and reboot) HRTs (and other radical kernels) alongside traditional kernels [15, 30]. Leaving our removal of the kernel/user distinction aside, the HRT model can be viewed as a form of Exokernel [8], and constructed HRTs are single-purpose, similar to unikernels [27], which are becoming accepted within the general cloud/data center space. Other forms of HPC unikernels are also emerging [12].

Gang scheduling. When time-sharing a distributed memory parallel machine, it is possible for the application to unnecessarily block on one node because it is descheduled on another. For example, a send from one node may block on a receive on another. The probability and performance impact of such unnecessary blocking increase dramatically as the application scales. Independent of this issue, It is also well known that collective communication can be more efficient if synchronization and data transfer can be decoupled as in classic direct deposit [38] or today's RDMA [24], but this implies receivers are running when senders deliver messages.

Gang scheduling, in which each time-shared node runs the application simultaneously, was proposed by Ousterhout in 1982 [32] to address these kinds of concerns. Classic gang scheduling (e.g. [18]) uses global synchronization to guarantee coordinated scheduling decisions on different nodes, and is perceived to be difficult to scale on some hardware. Implicit co-scheduling [7] avoids global coordination and instead coordinates node scheduler actions through inference of likely remote scheduler activity from local interactions. This produces gang scheduled behavior with high probability.

Gang scheduling of processes or threads within a single node was shown early on to allow faster synchronization mechanisms to be employed [9]. More recently, a similar idea has become important for virtual machines. Here, a particular concern is that a descheduled virtual CPU that is holding a spin lock can very inefficiently block other virtual CPUs that are attempting to acquire the spin lock [39]. Gang scheduling the virtual CPUs of a VM is one way to address the problem.

Our work provides guaranteed gang scheduled behavior of threads on a single node without global synchronization.

Real-time systems. Real-time systems research dates back to the earliest days of computing. Both soft and hard real-time systems introduce the concept of a deadline (or a generalization in the form of a utility function [35]) in scheduling tasks or threads, and the concept of making scheduling time-driven [17] as opposed to providing some notion of fairness as in commodity schedulers. In a hard real-time system, all deadlines must be met in order for the system to be correct [37]. The theory of hard real-time systems can be implemented in numerous ways, of which an RTOS is one. The cornerstone theoretical result is that of Liu and Layland [23], which we use as well. In our work, we also adopt the theoretical model of Liu [25] to integrate real-time and non-real-time scheduling in a single, per-CPU RTOS framework.

To the best of our knowledge, our work on VSched [21], which used a soft real-time model to allow time-sharing of single-node programs with controlled interference, was the first use of real-time systems in HPC. Subsequent work extended this model to distributed memory parallel machines. Similar to scheduling in Barrelfish, which we describe below, we showed how to use an infrequently invoked global feedback control loop to dynamically set real-time constraints on individual nodes so as to achieve coordinated scheduling (and the effect of gang scheduling) with minimal communication [6, 22]. This allowed an administrator to throttle up/down application resource usage with commensurate performance changes. In part, the current work extends these ideas to the node level using a hard real-time model.

Mondragon, et al [29] describe using an earliest-deadline-first (EDF) scheduler to time-share simulation and analysis codes with isolation, but this did not involve either hard real-time or coordinated scheduling.

Operating systems. Tesselation and Barrelfish are operating systems research projects of particular relevance to the present work. Tesselation [4, 26], implements a "space-time" partitioning model for the node's CPU resources, allowing associated threads to be scheduled simultaneously across a group of CPUs in a gang scheduled manner. The group scheduling component of our work achieves a similar effect. As far as we are aware, however, Tesselation does not use a hard real-time model to achieve its goals.

Barrelfish is an implementation of a multikernel [2]. Multikernels attempt to achieve better scalability on a multicore node (as well as to support a heterogeneous node) by adopting a largely distributed model in which per-CPU kernels strictly communicate using explicit messaging. In contrast, our HRT model shares all state across CPUs and between kernel and application, although we aggressively use traditional per-CPU state where necessary for performance. We also focus on a homogeneous node.

Scheduling in Barrelfish [33, 34] was heavily influenced by HPC scheduling, in particular gang scheduling. Similar to our design, Barrelfish uses per-CPU real-time schedulers whose clocks are synchronized. A gang scheduling abstraction called phase-locked scheduling is then implemented on top of these deterministic schedulers by selecting the same scheduling constraints for each member of the gang. Our work has several differences. First, we attempt to achieve hard real-time behavior on x64 despite SMIs through the use of an eager EDF scheduling model on the individual CPU. Second, we introduce interrupt steering and segregation to further enhance hard real-time behavior. Third, we incorporate lightweight tasks within our scheduling model. Fourth, we give a detailed performance evaluation of our system as a hard real-time system. Finally, we illustrate how barriers may be eliminated in some cases using very fine grain scheduling.

8 CONCLUSIONS AND FUTURE WORK

The fusion of hard real-time systems and parallel systems holds considerable promise, and this paper strongly supports the feasibility of such a fusion. We have described the design, implementation,

and performance of a hard real-time scheduler for parallel systems at the node level. The scheduler, which will be publicly available within the Nautilus codebase for x64 hardware, is able to achieve isolation and very fine resolution timing constraints, based on wall clock time, both for individual threads and groups of threads that execute synchronously. Our scheduler is able to run a fine grain BSP benchmark across a Xeon Phi such that its resource consumption can be cleanly throttled. The scheduler also makes it possible to replace barriers in the benchmark with scheduling behavior that is time-synchronized across CPUs.

We are currently working on adding real-time, and barrier removal support to Nautilus-internal implementations of OpenMP [31] and NESL [3] run-times. We are also exploring compiling parallel programs directly into cyclic executives, providing real-time behavior by static construction.

REFERENCES

[1] Rolf Riesen Arthur B. Maccabe, Kevin S. Mccurley and Stephen R. Wheat. 1994. SUNMOS for the Intel Paragon: A Brief User's Guide. In *Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference.* 245–251.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP).*

[3] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel and Distrib. Comput.* 21, 1 (April 1994), 4–14.

[4] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miguel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. 2013. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proceedings of the* 50^{th} *ACM/IEEE Design Automation Conference (DAC 2013).* 76:1–76:10.

[5] Brian Delgado and Karen Karavanic. 2013. Performance Implications of System Managemnet Mode. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC 2013).*

[6] Peter Dinda, Bin Lin, and Ananth Sundararaj. 2009. Methods and Systems for Time-Sharing Parallel Applications with Performance-Targetted Feedback-Controlled Real-Time Scheduling. (February 2009). United States Patent Application 11/832,142. Priority date 8/1/2007.

[7] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. 1996. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems.* 25–36.

[8] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. 1995. ExoKernel: An Operating System Architecture for Application Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating System Principles.* 256–266.

[9] Dror G. Feitelson and Larry Rudolph. 1992. Gang Scheduling Performance Benefits for Fine-grain Synchronization. *J. Parallel and Distrib. Comput.* 16, 4 (1992), 306–318.

[10] Kurt Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 ACM/IEEE conference on Supercomputing (SC).* 1–12.

[11] Alexandros V. Gerbessiotis and Leslie G. Valiant. 1994. Direct Bulk-Synchronous Parallel Algorithms. *J. Parallel and Distrib. Comput.* 22, 2 (1994), 251–267.

[12] Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Atsushi Hori, and Yutaka Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016).*

[13] Kyle Hale. 2016. *Hybrid Runtime Systems.* Ph.D. Dissertation. Northwestern University. Available as Technical Report NWU-EECS-16-12, Department of Electrical Engineering and Computer Science, Northwestern University.

[14] Kyle Hale and Peter Dinda. 2016. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016).*

[15] Kyle Hale, Conor Hetland, and Peter Dinda. 2017. Multiverse: Easy Conversion of Runtime Systems Into OS Kernels. In *Proceedings of the 14th International Conference on Autonomic Computing (ICAC 2017).*

[16] Kyle C. Hale and Peter A. Dinda. 2015. A Case for Transforming Parallel Runtime Systems Into Operating System Kernels (short paper). In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015).*

[17] E. Douglas Jensen, C Douglass Lock, and Hideyuki Tokuda. 1985. A Time-Driven Scheduling Model for Real-Time Operating Systems. In *Proceedings of the Real-Time Systems Symposium.* 112–122.

[18] Morris Jette. 1997. Performance characteristics of gang scheduling in multiprogrammed environments. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing.* 1–12.

[19] Brian Kocoloski, Leonardo Piga, Wei Huang, Indrani Paul, and John Lange. 2016. A Case for Criticality Models in Exascale Systems. In *Proceedings of the 18th IEEE International Conference on Cluster Computing (CLUSTER 2016).*

[20] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. 2010. Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing. In *Proceedings of the* 24^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010).*

[21] Bin Lin and Peter Dinda. 2005. Vsched: Mixing Batch and Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing).*

[22] Bin Lin, Ananth Sundararaj, and Peter Dinda. 2007. Time-sharing Parallel Applications With Performance Isolation And Control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC).* An extended version appears in the Journal of Cluster Computing, Volume 11, Number 3, September 2008.

[23] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM* 20, 1 (January 1973), 46–61.

[24] Jiunxing Liu, Jiesheng Wu, and Dhabaleswar Panda. 2004. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming* 32, 3 (June 2004), 167–198.

[25] Jane W. S. Liu. 2000. *Real-Time Systems.* Prentice Hall.

[26] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the* 1^{st} *USENIX Conference on Hot Topics in Parallelism (HotPar 2009).* 10:1–10:6.

[27] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the* 18^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013).*

[28] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Computing* 12, 10 (2001), 1094–1104.

[29] Oscar Mondragon, Patrick Bridges, and Terry Jones. 2015. Quantifying Scheduling Challenges for Exascale System Software. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2015).*

[30] Jiannan Oayang, Brian Kocoloski, John Lange, and Kevin Pedretti. 2015. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015).*

[31] OpenMP Architecture Review Board. 2008. *OpenMP Application Program Interface 3.0.* Technical Report. OpenMP Architecture Review Board.

[32] John Ousterhout. 1982. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Conference on Distributed Computing Systems (ICDCS).*

[33] Simon Peter. 2012. *Resource Management in a Multicore Operating System.* Ph.D. Dissertation. ETH Zurich. DISS.ETH NO. 20664.

[34] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. 2010. Design Principles for End-to-end Multicore Schedulers. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism (HotPar).*

[35] Raj Rajkumar, Chen Lee, John Lehoczky, and Dan Siewiorek. 1997. A Resource Allocation Model for QoS Management. In *Proceedings of the IEEE Real-Time Systems Symposium.*

[36] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. 2009. Designing and Implementing Lightweight Kernels for Capability Computing. *Concurrency and Computation: Practice and Experience* 21, 6 (April 2009), 793–817.

[37] John Stankovic and Krithi Ramamritham. 1988. *Hard Real-Time Systems.* IEEE Computer Society Press.

[38] Thomas Stricker, James Stichnoth, David O'Hallaron, Susan Hinrichs, and Thomas Gross. 1995. Decoupling Synchronization and Data Transfer in Message Passing Systems Of Parallel Computers. In *Proceedings of the International Conference on Supercomputing.* 1–10.

[39] Chuliang Weng, Qian Liu, Lei Yu, and Minglu Li. 2011. Dynamic Adaptive Scheduling for Virtual Machines. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC).*

[40] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. 1994. PUMA: An Operating System for Massively Parallel Systems. *Scientific Programming* 3 (1994), 275–288.