Pushing Software Events to the Hardware Limit

Kyle C. Hale and Peter Dinda

Abstract

Runtimes and applications that rely heavily on event notifications suffer when such notifications must traverse several layers of processing in software. Many of these layers necessarily exist in order to support a general-purpose, portable kernel architecture, but they introduce unacceptable overheads for demanding, high-performance parallel runtimes. Other overheads can arise out of a mismatched event programing or system call interface. Whatever the case may be, the average latency and variance in latency of commonly used software mechanisms for event notifications is abysmal compared to the hardware limit, which is several orders of magnitude lower.

One barrier to low-latency events is the user/kernel-mode distinction. Motivated by experience working with several parallel runtimes—and the limitations of their operation in user-space—we explore the limits of low-latency event notifications in an execution environment, the hybrid runtime (HRT), that eliminates the user/kernel distinction. We propose several mechanisms that employ kernel mode-only features to accelerate event notifications by up to 4,000 times and provide a detailed evaluation of our implementation using extensive microbenchmarks. Our evaluation is done both on a modern x64 server and the Intel Xeon Phi. Finally, we argue that a small addition to existing interrupt controllers (APICs) could push the limit of asynchronous events closer to the latency of the hardware cache coherence network.

Keywords: Software events, operating systems, Nemo, Hybrid Runtimes, runtime systems

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.

Pushing Software Events to the Hardware Limit

Kyle C. Hale and Peter A. Dinda
{k-hale, pdinda}@northwestern.edu
Department of Electrical Engineering and Computer Science
Northwestern University

ABSTRACT

Runtimes and applications that rely heavily on event notifications suffer when such notifications must traverse several layers of processing in software. Many of these layers necessarily exist in order to support a general-purpose, portable kernel architecture, but they introduce unacceptable overheads for demanding, high-performance parallel runtimes. Other overheads can arise out of a mismatched event programing or system call interface. Whatever the case may be, the average latency and variance in latency of commonly used software mechanisms for event notifications is abysmal compared to the hardware limit, which is several orders of magnitude lower.

One barrier to low-latency events is the user/kernel-mode distinction. Motivated by experience working with several parallel runtimes—and the limitations of their operation in user-space—we explore the limits of low-latency event notifications in an execution environment, the hybrid runtime (HRT), that eliminates the user/kernel distinction. We propose several mechanisms that employ kernel mode-only features to accelerate event notifications by up to 4,000 times and provide a detailed evaluation of our implementation using extensive microbenchmarks. Our evaluation is done both on a modern x64 server and the Intel Xeon Phi. Finally, we argue that a small addition to existing interrupt controllers (APICs) could push the limit of asynchronous events closer to the latency of the hardware cache coherence network.

1. INTRODUCTION

Many runtimes leverage event-based primitives as an out-of-band notification mechanism that can signal several things ranging from task completions or arrivals to message deliveries or changes in state. They may occur between logical entities like processes or threads, or they may happen at the hardware level. They are commonly used to build low-level synchronization primitives like mutexes or wait queues. Even the correct operation of a parallel program—whether the programmer is aware or not—in many cases relies crucially on incredibly low-latency event notifications traversing the CPU's cache coherence network.

Ultimately these events are just a special case of unidirectional, asynchronous communication, and so one might expect there to be little room for performance improvement. We have found, however, that exactly the opposite is true. While a cache-line invalidation can occur in a handful of CPU cycles and an inter-processor interrupt (IPI) can reach the opposite edge of a many-core chip in less than a thousand cycles, commonly used event signaling mechanisms like user-level condition variables fail to come within even three orders of magnitude of that mark.

We can perhaps explain some of this vast difference by appealing to feature creep and arguing that, however well designed, bloated operating system functionality will ultimately hinder performance. However, the sheer effort put into and success of highly tuned OSes like Linux limit the feasibility of this argument. Misuse or abuse of kernel interfaces by programmers might be a reasonable objection, but in our experience runtime developers typically have the sophistication necessary to extract every bit of performance out of the kernel and the available machine, so this scenario is also unlikely. If *neither* the implementation of the application or OS are to blame, how can we explain the poor performance of asynchronous software event notifications? In this paper, we argue that one of the biggest obstacles to approaching the hardware limit may actually be a fundamental issue with the *structure* of interactions between the application/runtime and the OS kernel, namely, the user-space/kernel-space distinction.

It has long been known that privilege transitions are expensive [17, 1]. AMD [2, Chapter 6.1.1] and Intel [21, Chapter 5.8.8] have both introduced new instructions into the ISA (e.g., `syscall/sysret`) to reduce the transition overhead but there are still several indirect effects that hinder performance such as TLB misses, cache misses, and various other software overheads associated with the kernel's design. Furthermore, such privilege short-circuiting instructions do not (currently) apply to events signaled asynchronously from a different hardware thread.

In HPC environments, where low-latency message delivery is critical, these transition overheads become even more detrimental. Furthermore, many HPC applications use structured communication patterns in which synchrony among nodes is paramount. Finally, nondeterminism caused by OS noise or the hardware on a single node (even a single CPU) will determine the performance of the application as a whole [11, 12, 18]. In response to these challenges and the perception among HPC application developers of operating systems "getting in the way", we have seen the emergence of lightweight kernels such as Kitten [25] and mOS [39], and hardware assists like RDMA for high-speed interconnects like Infiniband [20]. These systems observe the user-space/kernel-space distinction, however.

For applications and runtimes with strict performance requirements, the hybrid runtime (HRT) model presents a rich opportunity for deterministic performance and ultimate control over the machine. In this model, which we elaborate on in Section 2, the runtime (and application) essentially *is* the kernel, and determines the kernel abstractions it will use. An HRT has access to *all* of the hardware capabilities of the machine, and can thus leverage the hardware as necessary to achieve maximum performance. Not only is such access fully privileged, but there are also no privilege transitions.

The crux of this paper is to determine the hardware limits for asynchronous event notification on today's hardware, particularly on x64 NUMA machines and the Intel Xeon Phi, and then to achieve those limits with software abstractions implemented in the HRT model that can then be readily used by parallel runtimes ported to that model.

In the limit, an asynchronous event notification is bounded from below by the signaling latency on a hardware line. We measure and analyze inter-processor interrupts (IPIs) on our hardware, arguing that they serve as a first approximation for this lower bound. We consider both unicast and broadcast event notifications, which are used in extant runtimes, and have IPI equivalents. We then describe the design and implementation of *Nemo*, a system for asynchronous event notifications in HRTs that builds on IPIs. Nemo presents abstractions to the runtime developer that are identical to the pthreads condition variable unicast and broadcast mechanisms and thus are friendly to use, as well as much faster. Unlike IPIs, where a thread invokes an interrupt handler on a remote core, these abstractions are aimed at waking another thread on a remote core. In addition, Nemo provides an interface for unconventional event notification mechanisms that operate near the IPI limit.

As we show through a range of microbenchmarking on both platforms, Nemo is able to approach the hardware limit imposed by IPIs for the average latency and variance for asynchronous event notifications. Unicast notifications in Nemo have up to five times lower average latency than the user-level pthreads constructs, and the Linux futex construct, while broadcast notifications have up to 4,000 times lower average latency. Furthermore, the variance seen in Nemo is up to an order of magnitude lower for unicast, and many orders of magnitude lower for broadcast.

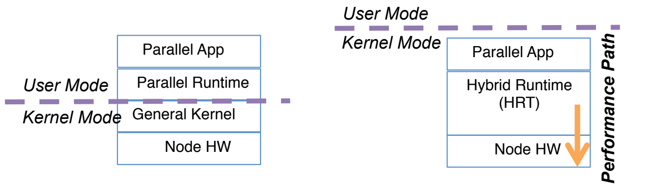
We then speculate about a small hardware change that would reduce the hardware limit (and Nemo's latency). Our measurements suggest that a large portion of IPI cost is due to the interrupt dispatch mechanism. The `syscall/sysret` instructions avoid similar costs for hardware thread-local system calls by avoiding this dispatch overhead. We propose that `syscall` be included as an IPI type. When receiving a "remote `syscall`" IPI, the faster dispatch mechanism would be used, reducing IPI costs for specific asynchronous events such as those in Nemo.

We make the following contributions:

- We outline the drawbacks of event notifications in user-space, analyzing the performance of event notifications and showing just how far they are from the performance that the hardware can provide.

- We show how the HRT model (in particular, moving event notifications into kernel-space) can alleviate some of these drawbacks by closely integrating with hardware capabilities.

- We present Nemo, an HRT-based event notification system which consists of three new notification mechanisms.



(a) Current Model (b) Hybrid Runtime Model

Figure 1: HRTs as compared to the existing model. The runtime and application now act as the kernel.

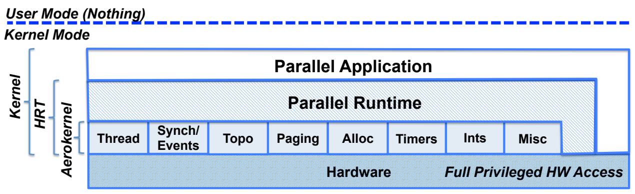


Figure 2: Structure of Aerokernel.

- We present an evaluation of Nemo's mechanisms using extensive microbenchmarks which show that they approach the hardware limits imposed by the IPI mechanism of the platform.

- We propose a further hardware assist for event notification in an HRT environment.

Nemo will be made available as a part of the open-source Aerokernel platform we describe in the next section.

2. HYBRID RUNTIMES AND AEROKERNEL

The hybrid runtime (HRT) model and the design and implementation of Aerokernel, our framework to support the model, has been driven by studying parallel runtimes including Legion [3], the NESL VCODE engine [4], the SWARM data flow runtime [26], ParalleX [22], Charm++ [23], the futures and places parallelism extensions to the Racket runtime [33, 34, 32], and nested data parallelism in Manticore [15, 14] and Haskell [5, 6]. In the case of Legion, SWARM, Racket, and a home-grown nested data parallel language, we also interviewed their developers to understand their views of the limitations of existing kernel support.

A common observation is that parallel runtimes often internally create abstractions and solve problems similar to those addressed by OS kernels. They do so without the advantage of running in kernel mode and are thus unable to leverage hardware functionality that could help. Additionally, parallel runtime developers typically perceive—often quite accurately—that the OS abstractions made available to them by general purpose or even lightweight kernels are poorly matched to their needs.

The HRT model promotes a parallel runtime to the same privilege as a kernel. In fact, the runtime (together with the application) acts *as* the kernel, enjoying access to the full capability set of the machine and ultimately determining the set of abstractions exposed to the application. Privilege transitions are unnecessary. This model is depicted in Figure 1.

To facilitate porting existing parallel runtimes to become HRTs, and to develop new HRTs from scratch, we developed Aerokernel, an open-source codebase of about 25,000 lines.

Aerokernel implements basic kernel functionality and building blocks that can be leveraged by HRT developers. In the style of libOS [10], the HRT or application may or may not choose to use these building blocks. They are simply offered for convenience. Nemo is one such building block. Aerokernel is linked with the runtime and application to form a full kernel that (currently) operates on x64 and Intel Xeon Phi hardware. Aerokernel eschews general purpose, non-performance-critical kernel features. These are delegated (on the Phi, to the host, and on x64, to a small subset of cores of the physical or virtual machine that run a general purpose kernel alongside the HRT). Figure 2 illustrates Aerokernel in the HRT context.

We have developed three HRTs based on Aerokernel, one that is a port of Legion, one that is a port of the NESL VCODE engine, and one for our home grown language's runtime. The HRT model can improve performance considerably. Sandia National Lab's HPCG application benchmark [9], which has been ported to Legion, is one example. Executing as an Aerokernel-based HRT, Legion HPCG operates up to 20% and 40% faster, for Xeon Phi and x64 (hardware as described in Section 3.1), respectively, than in its default user-level model.

3. EVENT NOTIFICATIONS AND THEIR LIMITS

Our motivation in exploring asynchronous event notifications in the HRT model stems from the observation that many parallel runtimes use expensive, user-level software events even though low-latency event communication has been extant in hardware for many years. However, these hardware capabilities are traditionally reserved for kernel-only use. We discuss common usage of event notification mechanisms, particularly for task invocations in parallel runtimes, then present measurements on modern multi-core machines for common event-based primitives, demonstrating potential benefits of the HRT model for low-latency events. Our core question for this section is just how fast *could* asynchronous event notification in current x64 and Phi hardware go.

3.1 Testbeds and measurement

We carried out all measurements in this paper on two machines. *x64* is a large x86_64 node, similar to what a supercomputer node might look like. It is a 2.1GHz AMD Opteron 6272 (Interlagos) server machine with 64 cores and 128 GB of memory spread out across four sockets and eight NUMA nodes. All CPU cores in a single NUMA node share an L3 cache, and within the NUMA nodes, CPUs are grouped into four pairs of hardware threads. Hardware threads (hyperthreads) share an L1 i-cache and a unified L2 cache. Each hardware thread has its own L1 d-cache. This machine is configured for "Max performance" in the BIOS to eliminate power management effects on measurement. It also has a "freerunning" TSC, which means that the TSC will tick at a constant rate regardless of the core frequency. For Linux tests, it runs Red Hat 6.5 with stock Linux kernel version 2.6.32. *Phi* is an actively cooled Intel Xeon Phi 3120A PCI accelerator. Our card is set up with the Intel MPSS 3.4.2 toolchain and the stock Linux μOS, which is based on kernel version 2.6.38.

Time measurement in both cases is with the cycle counter and measurements are taken over at least 1000 runs (unless otherwise noted) with results shown as box plots or CDFs and summary statistics overlaid in some cases.

3.2 Runtime events

In one common usage pattern of asynchronous event notifications, a signaling thread is able to notify one or more waiting threads that they should continue. The signaling thread continues executing regardless of the status of waiting threads.

In examining the usage of event notifications, we worked with several modern parallel runtimes, including Charm++ [23], SWARM [26], and Legion [3, 36]. They all use asynchronous events in some way, whether explicitly through an event programming interface or implicitly by runtime design. In many cases, these runtimes use events as vehicles to notify a remote worker that there is work to do or that it should execute a particular task.

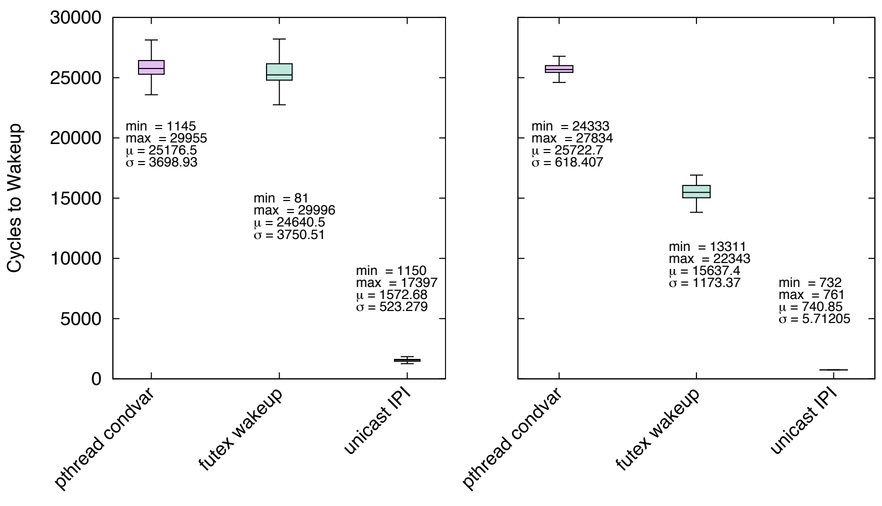
Legion provides a good example. Legion has a logical processor model in which a thread (e.g., a pthread) implements a logical processor. Each logical processor sequentially operates over tasks. In order to notify remote logical processors that a task is ready to execute, the signalling processor broadcasts on a condition variable (e.g., a `pthread_cond_t`) that wakes up any idle logical processors, all of which race to claim the task for execution. This is not entirely different from the `schedule()` interrupts used in Linux at the kernel level. Since `pthread_cond_broadcast()` must involve the kernel scheduler (via a system call), it is fairly expensive, as we will shown in Section 3.3. Linux's futex abstraction attempts to ameliorate this cost with mixed success.

3.3 Microbechmarks

Figure 3 shows the latency for event wakeups on *x64* and *phi*. In each of these experiments, we create a thread on a remote core. This thread goes to sleep until it receives an event notification. We measure the time from the instant before the signalling thread sends its notification to when the remote thread awakens. The threads are mapped to distinct cores. The numbers represent statistics computed over 100 trials for each remote core (6,300 trials on x64, 22,700 on phi).

Three mechanisms are compared. The first two are the most commonly used asynchronous event mechanisms in user-space: condition variables and futexes. The pthreads implementation of condition variables depicted is built on top of futexes. We can see that the overhead of condition variables compared to futexes may be significant, but it is platform dependent or implementation dependent—the average event wakeup latency of condition variables is nearly double that of futexes on the Phi, but only a small increment more on x64.

The third mechanism, denoted with "unicast IPI" on the figure, shows the unicast latency of an inter-processor interrupt (IPI). On *x64* and *phi*, each hardware thread has an associated interrupt controller (an APIC). The APIC commonly *receives* external interrupts and initiates their delivery to the hardware thread, but it is also exposed as a memory-mapped I/O device to the hardware thread. From this interface, the hardware thread can program the APIC to *send* an interrupt (an IPI) to one or more APICs in the system. APICs are privileged devices and typically used only by the kernel.



(a) x64 (b) phi

Figure 3: Comparing existing unicast event wakeups in user-mode (pthreads and futexes on Linux) with IPIs on *x64* and *phi*. Unicast IPIs are at least an order of magnitude faster and exhibit much less variation in performance on both platforms.

As we can see from Figure 3, IPIs are on average much faster than either condition variables or futexes. On *x64*, they have roughly 16 times lower latency than either, while on *phi*, they have roughly 32 times lower latency than condition variables and 16 times lower latency than futexes. On *phi*, the average IPI latency is only 700 cycles. The wall-clock time on the two machines is similar, as *x64* has roughly twice the clock rate.

It is important to note that IPIs are not doing the same thing as a condition variable or a futex. For IPIs, we measure the time from the instant before the signaling thread sends its notification to when the *interrupt handler* begins executing, not the waiting thread. We are interested in the IPI time because this measurement represents a lower bound for a wakeup mechanism using existing hardware functionality on commodity machines. There is significant room for an improvement of more than an order of magnitude (∼20x). We will attempt to achieve this improvement in Section 4.

It also important to observe that not only is the average time much lower in an IPI, but its variance is also diminished considerably. As we noted in the introduction, variance in performance limits parallel runtime performance and scalability. This is an OS noise problem. The hardware has a lot less of it.

Broadcast events are of significant interest in parallel runtimes, for example in Legion as described above. Figure 4 shows the wakeup latency for a broadcast event on *x64* and *phi*, again comparing a condition variable based approach in pthreads, Linux futex, and IPIs. Measurements here operate as with the unicast events, but we keep track of time of delivery on every destination as well so we can assess how synchronous the deliveries are.

We can see that the relative latency improvements for broadcasts with condition variables and futexes are similar to the unicast case, but the gain of using broadcast IPIs is much larger. On *phi*, the average latency of a broadcast IPI being received by all targets is over 4,000 times lower than for a condition variable solution. The gain in variance

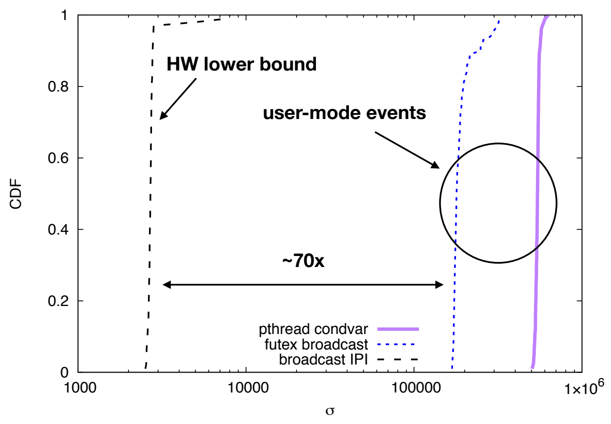
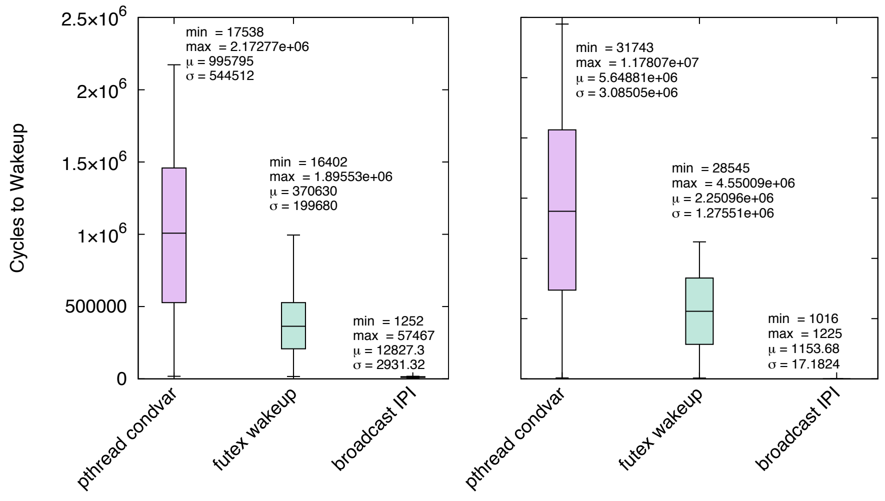


Figure 5: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *x64*. Broadcast IPIs achieve a synchrony that is three orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

is similarly startling. On *x64*, this gain is 78 times. While broadcast IPIs exploit the hardware's own parallelism, the implementations of all the other techniques are essentially serialized in a loop that wakes up waiting threads sequentially. In part this difference between *phi* and *x64* is simply that the Phi has almost four times as many cores. While one could argue that a programmer should use a barrier over a condition variable for a wakeup on many cores, barriers lack the asynchrony needed for these kinds of event notifications.

We should also hope that a broadcast event causes wake-ups to occur across cores with synchrony; when a broadcast event is signaled, we would like all recipients to awaken as close to simultaneously as possible. However, Figures 5 and 6 show that this is clearly not the case for the condition



(a) x64 (b) phi

Figure 4: Comparing existing user-space event broadcasts vs. IPIs on *x64* and *phi*. Broadcast IPIs have over 4000 times lower latency than condition variables and almost 2000 times lower latency than futexes on *phi*. *x64* shows 78 times lower latency than condition variables and 30 times lower altency than futexes. Variance in latency is similarly reduced.

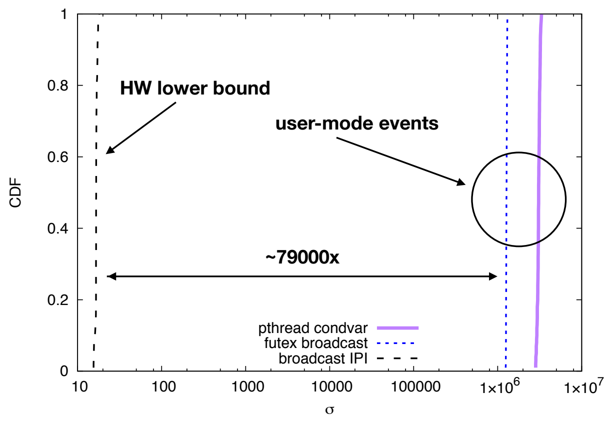


Figure 6: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *phi*. Broadcast IPIs achieve a synchrony that is five orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

variable or futex-based broadcasts. Recall that we measure the time of the wakeup on each destination. For one broadcast wakeup, we thus have as many measurements as there are cores, and we can compute the standard deviation (σ) among them. In these figures, we repeat this many times and plot the CDFs of these σ estimates. Note that in the figures the x-axes are on a log scale. On these platforms, there are several orders of magnitude difference in the degree of synchrony in wakeups achievable on the hardware and what is actually achieved by the user-space mechanisms.

3.4 Discussion

The tremendous gap between the performance of asynchronous software events in user-mode and the hardware capabilities should cause concern for runtime developers. Not only do these latencies indicate that software wakeups may happen roughly on the millisecond time-scale of an extremely slow network packet delivery, but also that the programmer can do very little to ensure that these wakeups occur with *predictable* performance. The problem is much worse for broadcast events, and *the problem* appears to scale with increasing core count.

Recall again that we claim IPIs are a hardware limit to event notifications, and that it is important to understand that an IPI is not an event notification by itself. The goal of Nemo is to achieve event notifications compatible with those expected by parallel runtimes with performance that approaches that of IPIs, as well as to offer unconventional mechanisms that tradeoff ease of use for performance near the IPI limit.

4. NEMO EVENTS

Nemo is an asynchronous event notification system for HRTs that is built within our Aerokernel framework. Nemo addresses the terrible performance of asynchronous user-space notifications by leveraging hardware features not commonly available to runtime or application developers. That is, they are enabled by the fact that the entire HRT runs in kernel mode.

The goal of Nemo is to approach the hardware IPI latency profile. Figure 7 represents in detail the kind of profile we would like to achieve. We expect that these numbers, which were measured on *x64*, will tell us something about the machine, given its complex organization. The knees in the curve (marked with black circles) indicate boundaries in the IPI network. While we could not find reliable documentation from AMD or other parties on the topology of the interprocessor interrupt network on this machine, we are fairly confident that these inflection points correspond to distances within the chip hierarchy as indicated in the captions. As Nemo begins to exhibit similar behavior, we will know we

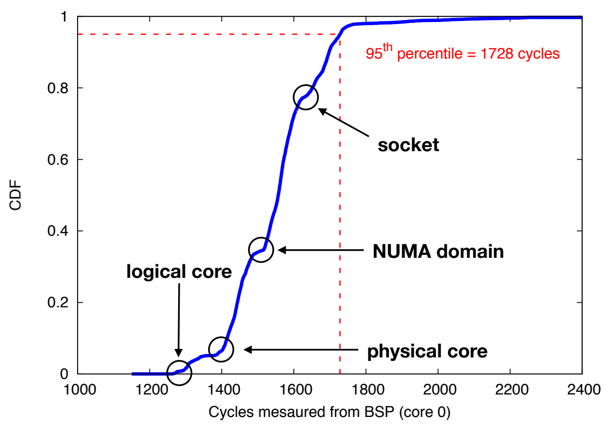


Figure 7: CDF of unicast IPI latency from the BSP to all other cores on *x64*. Approaching this profile is the goal of Nemo.

are near the limits of available hardware.

Unicast IPI latencies on our *phi* card (not shown) are smaller and show less pronounced inflection points. We suspect this is due to the fact that it is a single-chip processor with a fairly balanced interconnect joining the cores.

4.1 Kernel-mode condition variables

Existing runtimes, such as Legion, use pthreads features in their user-space incarnations. Aerokernel tries to simplify the porting of such runtimes to become HRTs. To support thread creation, binding, context switching, and similar elements, Aerokernel provides a pthreads-like interface for its kernel threads. Default thread scheduling (round-robin with or without preemption—preemption is not used here) and mapping policies (initial placement by creator, no migration) are intended to be simple to reason about. Similarly, memory allocation is NUMA-aware and based on the calling thread's location, not by the first touch.

For Nemo, the relevant event mechanism in pthreads is the condition variable, implemented in the `pthread_cond_X()` family of functions. Nemo implements a compatible set of these functions within Aerokernel. There are two implementations. In the first, there is no special interaction with the scheduler. When a waiting thread goes to sleep on a condition variable, it puts itself on the condition variable's queue and deschedules itself. When a signaling thread invokes `condvar_signal()`, this function will put the waiting thread back on the appropriate processor's ready queue. The now signalled thread will not run until the processor's background thread `yield()`s. We would expect this implementation to increase performance simply by eliminating user/kernel transitions from system calls, e.g. the `futex()` system call.

The second implementation uses a more sophisticated interaction with the scheduler in order to better support common uses in runtimes like Legion and SWARM. In these, the threads that are sleeping on condition variables are essentially logical processors. Ideally each one would map to a single physical CPU and would not compete for resources on that CPU. Scheduling of tasks (Legion tasks or SWARM tasks, not kernel threads) are handled by the runtime, so kernel-level scheduling is superfluous. The condition vari-

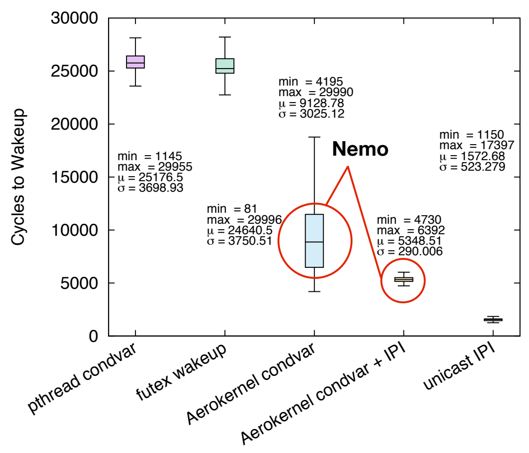


Figure 8: Nemo kernel-mode event mechanisms for single wakeups on *x64*. Average latency is reduced by over a factor of four, and variation is considerably reduced.

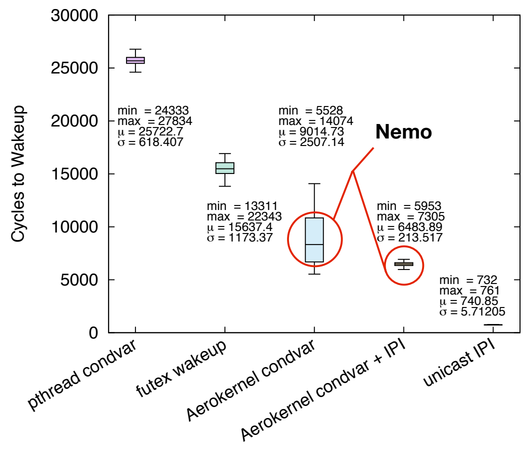


Figure 9: Nemo kernel-mode event mechanisms for single wakeups on *phi*. Average latency is reduced by over a factor of four and variation is considerably reduced.

able in such systems is used essentially to awaken logical processors.

On a `condvar_signal()` our second implementation sends an IPI to "kick" the physical processor of the newly runnable thread. The scheduler on the physical processor can then immediately switch to it. The kick serves to synchronize the scheduling of the sleeping thread, reducing the effects of background threads that may be running.

The two implementations comprise about 200 lines of code within the Aerokernel framework.

Figures 8 and 9 show the performance of these two implementations compared to the existing user-space techniques and to the unicast IPI. Our first implementation ("Aerokernel condvar") roughly halves the median latency of user-mode event wakeups on both *x64* and *phi*. This latency improvement represents a rough estimate of the speedup achieved solely by moving the application/runtime into kernel-mode, thus avoiding kernel/user transition overheads. The implementation does, however, exhibit fairly large variance

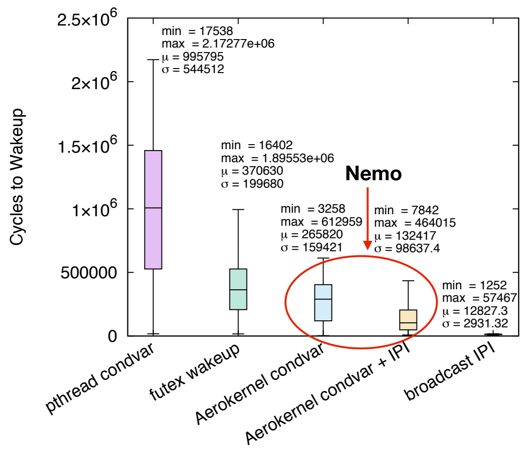


Figure 10: Nemo kernel-mode event mechanisms for broadcast wakeups on *x64*. Average latency is reduced by a factor of 10 and variation is considerably reduced.

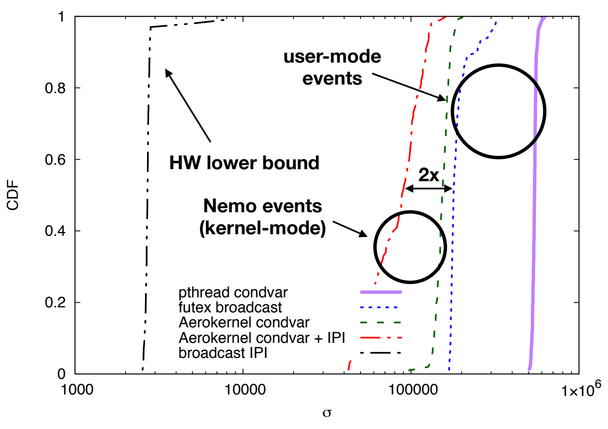


Figure 12: CDF showing the Nemo kernel-mode event mechanisms for broadcast wakeups on *x64*. Nemo is able to achieve an order of magnitude better synchrony in thread wakeups.

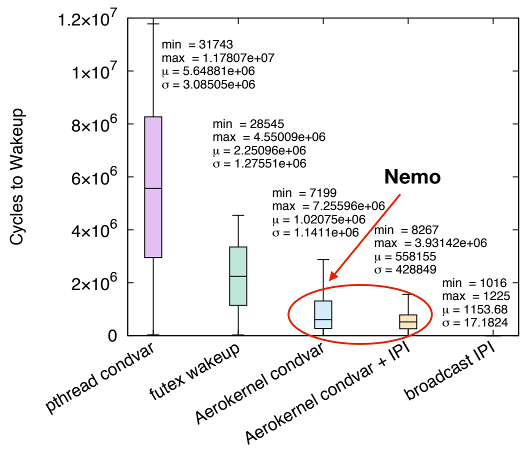


Figure 11: Nemo kernel-mode event mechanisms for broadcast wakeups on *phi*. Average latency is reduced by an factor of 16 and variation is considerably reduced.

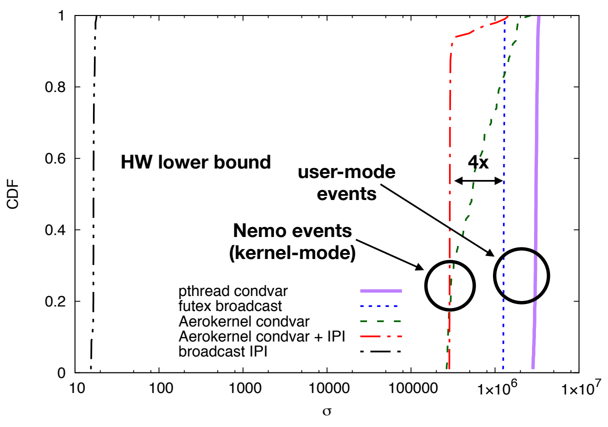


Figure 13: CDF showing the Nemo kernel-mode event mechanisms for broadcast wakeups *phi*. Nemo is able to achieve an order of magnitude better synchrony in thread wakeups.

in wakeup latency. This is because the wakeup time depends on how long it takes for the CPU's background thread to `yield()` again.

Our second implementation ("Aerokernel condvar + IPI") ameliorates this variation, and further reduces average and median latency. The use of the IPI kick collapses the median latency of the wakeup down to the minimum latency of the standard kernel-mode condition variable. The variance in this case is much lower than all of the other wakeup mechanisms.

Figures 10 and 11 show the performance of broadcast events, where the gain is larger (a factor of 10–16). Figures 12 and 13 show the improvement of the synchrony of broadcast event wakeups. This is improved by a factor of 10 on both platforms. Section 3.3 gives a description of the format of the latter two figures and a discussion of broadcast IPIs.

In the current Nemo implementations for broadcast events, the signaling thread moves each waiting thread to its processor's run queue and then (in the second implementation) kicks that processor with an IPI. Although there is considerable overlap between context switches, in-flight IPIs, and moving the next thread to its run queue, we expect that this sequential behavior of the signaling thread is a current limit on the broadcast event mechanism both in terms of average/median latency and in terms of synchrony of the awakened threads. This is in contrast to the IPI broadcast in hardware, which is inherently parallel and exhibits significant synchrony in arrivals, as can be seen from the figures and previous discussions.

4.2 IPIs and Active Messages

In the previous section, we introduced Nemo events which were built to conform to the pthreads programming interface, particularly condition variables. With the inherent limitations of this interface and the privileged hardware available to us in an HRT in mind, we now explore a new event mechanism with a different interface that is built directly on top of IPIs. The mechanism is also informed by how

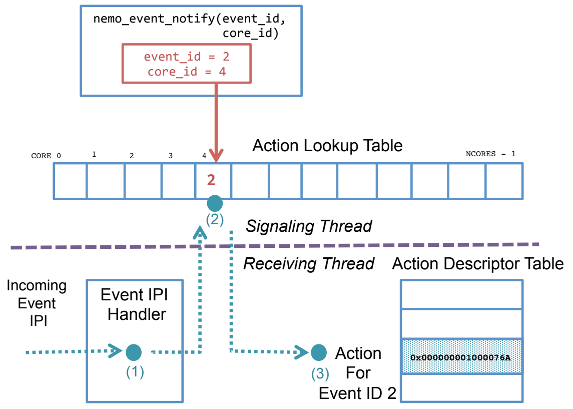


Figure 14: Nemo Active Messages-inspired event wakeups implemented using IPIs.

pthreads condition variables are actually used in Legion and SWARM, namely to *indirectly* implement behavior via user-level mechanisms that can be *directly* implemented in the kernel context.

We claim that Active Messages [37] would be better fit to the functional behavior that many runtimes need. Active Messages were introduced to enable extremely low-latency message handling for distributed memory supercomputers with high-performance interconnects. Since a message delivery is ultimately just one kind of asynchronous event, we looked to Active Messages for inspiration on how to approach the hardware limit for asynchronous software events. In short, we use the IPI as the basis for an Active Message model within the shared memory node.

In an Active Message system, the message payload includes a reference (a pointer) to a piece of code on the destination that should handle the message receipt. One advantage of this model is that it reduces the load on the kernel and results in a faster delivery to the user-space application. Since the HRT *is* the kernel, we do not need to avoid transferring control to it on an event notification. We can eliminate handling overhead by leveraging existing logic in the hardware already meant for handling asynchronous events—in this case, IPIs. IPIs are not a complete Active Message substrate, however, as there is no payload other than the interrupt vector.

Figure 14 shows the design and control flow of our Active Messages-inspired event mechanism. We reserve a slot in the IDT for a special Nemo event interrupt, which will vector to a common handler (1). If only one type of event is necessary, this handler will be the final handler and thus no more overhead is incurred. However, it is likely that a runtime developer will need to use more than one event. In this case, the common handler will lookup an event *action* (a second-level handler) in an *Action Lookup Table* (ALT), which is indexed by its core id (2). From this table, we find an *action ID*, which serves as an index into a second table called the *Action Descriptor Table* (ADT). The ADT holds actions that correspond to events. After the top-level handler indexes this table, it then executes the final handler (3). The IPI is used to deliver the active message, while the Action Table effectively contains its content.

The mechanism described here comprises about 160 lines of code in the Aerokernel context.

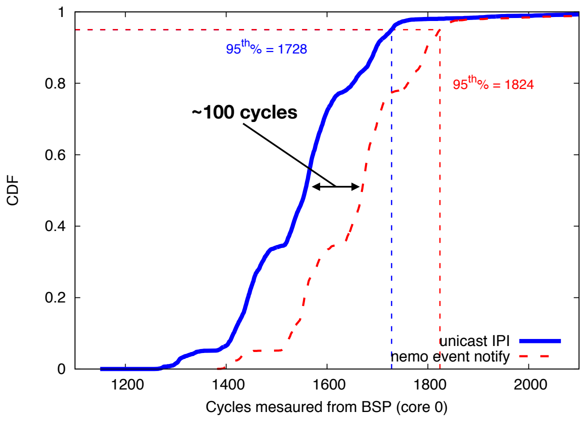


Figure 15: CDF showing unicast latency of Nemo's Active Messages-inspired events compared to unicast IPIs on *x64*. Nemo is within ∼5% of IPI performance.

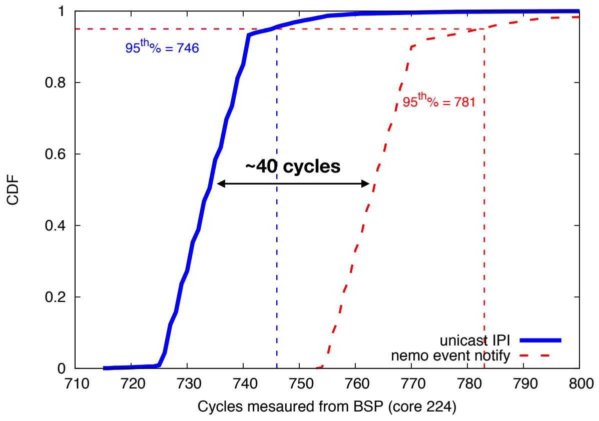


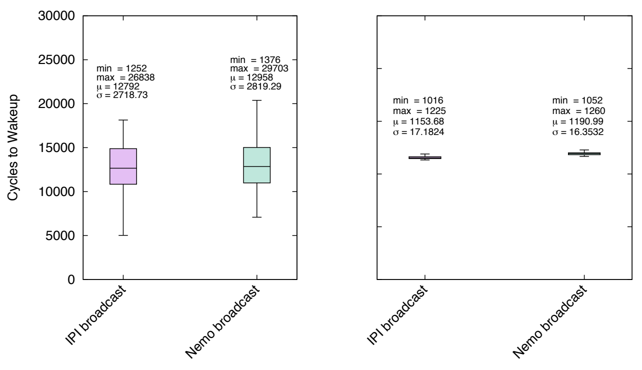
Figure 16: CDF showing unicast latency of Nemo's Active Messages-inspired events compared to unicast IPIs on *phi*. Nemo is within ∼5% of IPI performance.

Figures 15 and 16 show CDFs of the latency of Nemo's Active Messages-inspired events compared to the unicast IPI. Notice that in all cases Nemo events are only roughly 40 cycles slower on *phi* and 100 cycles slower on *x64*. We are now truly close to the capabilities of the hardware as evidenced by the performance and by the observed sensitivity to the hardware topology, which is implied by the knees in the IPI latency profile (e.g. in Figure 7).

Figure 17 shows the latency of Active Messages-inspired Nemo events compared to broadcast IPIs. The performance of the Nemo events are within a few tens of cycles of broadcast IPIs, as we would expect. Figures 18 and 19 show the amount of synchrony present in the Nemo events. An explanation of this form of figure is given in Section 3.3.

5. TOWARDS IMPROVING THE HARDWARE LIMIT

Although we have shown a marked improvement for asynchronous software events using hardware features (IPIs in particular) that are not commonly available to user-space



(a) x64 (b) phi

Figure 17: Broadcast latency of Nemo's Active Messages-inspired events compared to broadcast IPIs on *x64* and *phi*. Performance is nearly identical in both cases.

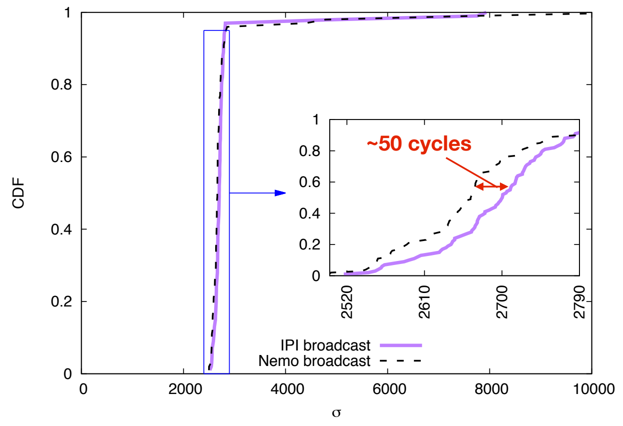


Figure 18: CDF comparing Nemo's Active Messages-inspired broadcast events compared to broadcast IPIs on x64. Synchrony is nearly identical.

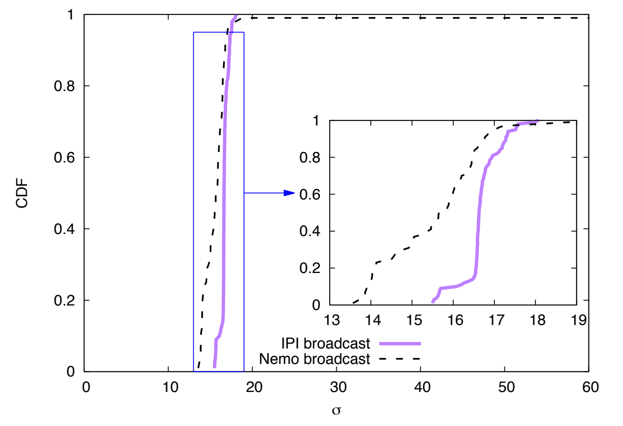


Figure 19: CDF comparing Nemo's Active Messages-inspired broadcast events compared to broadcast IPIs on phi. Synchrony is nearly identical.

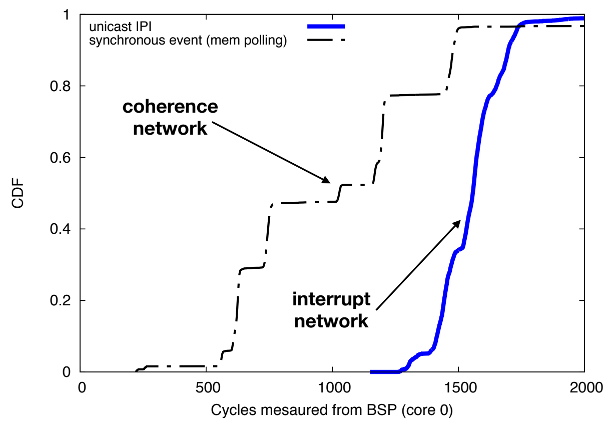


Figure 20: CDF comparing latency of asynchronous unicast IPIs compared to a simple synchronous notification scheme using memory polling on x64. This represents the basic cost difference between a synchronous and asynchronous event imposed by the hardware.

programs, we would like to calibrate this performance to another hardware capability that is critical to the performance of multicore machines—the cache coherence network. The question we ask here is why can the hardware limit not be even better?

Mogul et al. lamented this issue [28] while advocating for lightweight, inter-core notifications: "Unfortunately, today IPIs are the only option."

The coherence network in a modern CPU propagates its own form of events between chips, namely messages that implement the protocol that maintains the coherence model. Not only do we expect the coherence network connecting the chips and the associated logic to be extremely low-latency but also very predictable in its performance.

How fast is this network from the perspective of event notification in general? We implemented a small *synchronous* event mechanism using memory polling to assess this. In this mechanism, much like in a barrier or a spinlock, the waiting thread simply spins on a memory location waiting for its value to change. When a signalling thread changes this value, its core's cache logic will send a coherence message to the waiting thread's core, ultimately prompting a cache fill with the newly written value, and an exit from the spin. Figure 20 shows the performance of this synchronous mechanism compared to the asynchronous mechanism of unicast IPIs on our x64 hardware. We can see that IPIs are about 1000 cycles more expensive until the notifications (or invalidations) have to travel further through the chip hierarchy and off chip. The stepwise nature of the "coherence network" curve confirms our prediction of predictable, low-latency performance.

These results prompted us to ask a new question: what is keeping the IPI network from achieving performance comparable to the coherence network? To address this question, we performed an analysis of IPIs from the kernel programmer's perspective, gathering measurements for the hardware and software events necessary for their delivery. Figure 21 shows the results.

The latency of a unicast IPI involves three components. The first, "Source APIC write", is the time to initiate the IPI

Software event	**min. cycles**
Source APIC write	43
Destination handling	729
Time on wire	378
Total for unicast IPI	**1150**
Total for `syscall`	**232**

Figure 21: Estimated IPI cost breakdown in cycles on *x64*.

by writing the APIC registers appropriately at the source. In the figure, we record the minimum time we observed. The second component, denoted "Destination handling", is the time required at the destination to handle the interrupt, going from message delivery to the time of the first interrupt handler instruction. To estimate this number, we measured the minimum latency from initiating a software interrupt (via an `int` instruction) to the entry of its handler on the same core. We expect that this number is actually an underestimate since it does not include any latency that might be introduced by processing in the destination APIC. The time on the wire is simply these two numbers subtracted from the total unicast IPI cost shown in Section 3.3. It is likely to be an overestimate.

Integrating the observations of Figures 20 and 21 suggests that the reason why an asynchronous IPI has so much higher latency than a synchronous coherence event is likely to be in large part due to the destination handling costs of an IPI. For asynchronous event notification in an HRT, much of this handling is probably not needed—we would like to simply invoke a remote function, much like a Startup IPI (SIPI) available on modern x86 machines. In particular, the privilege checks, potential stack switches, and stack pushes involved in an IPI are not needed.

A similar overhead was addressed a decade ago when it was shown how much overhead was involved in processing of the `int` instruction used in system calls, especially as clock speeds grew disproportionally to interrupt handling logic. Designers at Intel and AMD introduced the `syscall` and `sysret` instructions to reduce this overhead considerably. Figure 21 notes the cost of a `syscall` on our x64 hardware, which is less than 1/3 of our estimated destination handling costs for an IPI.

We believe similar benefits could be gained for low-latency event delivery and handling with a similar modification to the architecture. The essential concept is to introduce a new class of IPIs, the "remote `syscall`". This would combine the IPI generation and transmission logic with the `syscall` logic on the destination core. That is, this form of IPI would act like a `syscall` to the remote core, avoiding privilege checks, stack switches, or any stack accesses. To estimate the gains from this model, we made a projection of IPI performance if one could reduce destination handling to the cost of a `syscall` instruction. The projected improvements are shown in Figure 22. There is now considerable overlap in the performance of synchronous events based on the coherence network and asynchronous events based on the new "remote `syscall`".

Current Intel APICs use an Interrupt Command Register (ICR), to initiate IPIs. The delivery mode field, which is 3 bits long, indicates what kind of interrupt should be delivered. Mode `011` is currently reserved, so this is a possible candidate for a *remote* `syscall` mode. There are, of course

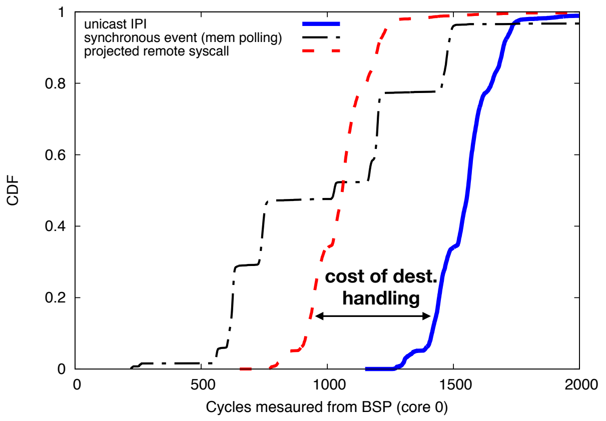


Figure 22: CDF of the projected latency of the proposed "remote syscall" mechanism. Comparison is made to the measured latencies of Figure 20.

numerous varieties of the APIC model between Intel and AMD, but the ICR is a 64 bit register with numerous reserved bits in all of them. Any of these bits could be used to encode a request for a "remote `syscall`". As another example, the 2 bit wide delivery shorthand field could be extended into the adjacent reserved field by one bit to accommodate indicating whether delivery should happen by the traditional interrupt mechanism or via `syscall`-like handling. In these delivery modes or shorthands, the vector might provide a hint to the event handling dispatch software. We expect that these changes would be fairly minimal, although we do not know what effort would be needed to integrate this new functionality with instruction fetch logic. The fact that a SIPI can already vector the core to a specific instruction suggests to us that it might not introduce much new logic. Indeed, another possible approach might be to allow SIPIs when the core is outside of its INIT state.

6. RELATED WORK

The real-time OS community has studied asynchronous events in depth, focusing on events in contexts such as predictable interrupt handling [30], priority inversion [31], and fast vectoring to user-level code [16]. However, there has yet been little discussion of these events as they apply to parallel runtimes, which have come into prominence more recently.

Thekkath and Levy introduced a mechanism [35] to implement *user-level* exception handlers—instances of synchronous events in our terminology—to mitigate the costs of the privilege transitions we discussed in Section 1. The motivation for this technique is not unlike that used for the RDMA-based techniques we see used in practice today.

Horowitz introduced a programmable hardware technique for *Informing Memory Operations*, which vector to a user-level handler with extremely low latency on cache miss events [19].

Keckler et al. introduced *concurrent event handling*, in which a multithreaded processor reserves slots for event handling in order to reduce overheads incurred from thread context switching [24]. Chaterjee discusses further details of this technique, particularly as it applies to MIT's J-Machine [29] and M-Machine [13].

The Message Driven Processor (MDP), from which the

J-Machine was built, had hardware contexts specifically devoted to handling message arrivals [8, 7]. Furthermore, this processor had an instruction (`EXECUTE`) that could explicitly invoke an action on a remote node. This action could be a memory dereference, a call to a function, or a read/write to memory. This is essentially the capability that in Section 5 we suggested could be implemented in the context of x64 hardware. It is unfortunate that useful explicit messaging facilities like those used in the MDP—save some emerging and experimental hardware from Tilera (née RAW [27]) and the RAMP project [38]—have not made their way into commodity processors used in today's supercomputers, servers, and accelerators.

7. CONCLUSIONS AND FUTURE WORK

We have shown how the performance of asynchronous software events is tremendously hindered by the existing model in which the application/runtime is restricted to user-space mechanisms and mismatched event programming interfaces. The performance of these mechanisms is nowhere near the hardware limits of IPIs, much less cache coherence. By moving the runtime into kernel-mode, creating what we call hybrid runtimes or HRTs, we could increase the performance of these event mechanisms considerably by implementing them directly on top of fully privileged hardware access. We did so by designing, implementing, and evaluating the Nemo asynchronous event system within our Aerokernel framework for building HRTs on x64 and Xeon Phi. HRTs built using Nemo primitives can enjoy event wakeup latencies that are as much as 4,000 times lower than the event mechanisms typically used in user-space. Furthermore, the variation in wakeup latencies in Nemo is much lower, and broadcast events cause wakeups that are much more simultaneous across cores. We then stepped back and considered the design of IPIs themselves and proposed a small hardware addition that could potentially reduce their cost considerably for constrained use cases, such as asynchronous event notification, thus pushing it closer to the performance of the hardware cache coherence network.

We next plan to evaluate the performance effects of these new mechanisms on existing parallel runtimes (as HRTs) and the difficulty of adapting these runtimes to Nemo.

8. REFERENCES

[1] How to speed up system calls. http://lwn.net/Articles/18411/. Accessed: 2015-08-11.

[2] AMD Corporation. *AMD64 Architecture Programmer's Manual Volume 2: Systems Programming*, May 2013.

[3] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. Legion: Expressing locality and independence with logical regions. In *Proceedings of Supercomputing (SC 2012)* (Nov. 2012).

[4] Blelloch, G. E., Chatterjee, S., Hardwick, J., Sipelstein, J., and Zagha, M. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing 21*, 1 (Apr. 1994), 4–14.

[5] Chakravarty, M., Keller, G., Leshchinskiy, R., and Pfannenstiel, W. Nepal—nested data-parallelism in haskell. In *Proceedings of the 7th International Euro-Par Conference (EUROPAR)* (Aug. 2001).

[6] Chakravarty, M., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. Data parallel haskell: A status report. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming* (Jan. 2007).

[7] Dally, W. J., Chao, L., Chien, A., Hassoun, S., Horwat, W., Kaplan, J., Song, P., Totty, B., and Wills, S. Architecture of a message-driven processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA 1998)* (June 1998), pp. 337–344.

[8] Dally, W. J., Davison, R., Fiske, J. S., Fyler, G., Keen, J. S., Lethin, R. A., Noakes, M., and Nuth, P. R. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro 12*, 2 (Apr. 1992), 23–39.

[9] Dongarra, J., and Heroux, M. A. Toward a new metric for ranking high performance computing systems. Tech. Rep. SAND2013-4744, University of Tennessee and Sandia National Laboratories, June 2013.

[10] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995)* (Dec. 1995), pp. 251–266.

[11] Ferreira, K. B., Bridges, P., and Brightwell, R. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of Supercomputing (SC 2008)* (Nov. 2008).

[12] Ferreira, K. B., Bridges, P. G., Brightwell, R., and Pedretti, K. T. Impact of system design parameters on application noise sensitivity. *Journal of Cluster Computing 16*, 1 (Mar. 2013).

[13] Fillo, M., Keckler, S. W., Dally, W. J., P., C. N., Chang, A., Gurevich, Y., and Lee, W. S. The M-Machine multicomputer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO 29)* (Nov. 1995), pp. 146–156.

[14] Fluet, M., Rainey, M., Reppy, J., and Shaw, A. Implicitly threaded parallelism in manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (Sept. 2008).

[15] Fluet, M., Rainey, M., Reppy, J., Shaw, A., and Xiao, Y. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming* (January 2007).

[16] Fry, G., and West, R. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In *Proceedings of the 2007 International Conference on Embedded Systems and Applications (ESA 2007)* (June 2007).

[17] Hayward, M. Intel p6 vs p7 system call performance. http://lwn.net/Articles/18412/. Accessed: 2015-08-11.

[18] Hoefler, T., Schneider, T., and Lumsdaine, A. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of Supercomputing (SC 2010)* (Nov. 2010).

[19] Horowitz, M., Martonosi, M., Mowry, T. C., and Smith, M. D. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA 1996)* (May 1996), pp. 260–270.

[20] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[21] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, Feb. 2014.

[22] Kaiser, H., Brodowicz, M., and Sterling, T. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the 38th International Conference on Parallel Processing Workshops (ICPPW 2009)* (Sept. 2009), pp. 394–401.

[23] Kalé, L. V., Ramkumar, B., Sinha, A., and Gursoy, A. The Charm parallel programming language and system: Part II–the runtime system. Tech. Rep. 95-03, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, 1994.

[24] Keckler, S. W., Chang, A., Lee, W. S., Chatterjee, S., and Dally, W. J. Concurrent event handling through multithreading. *IEEE Transactions on Computers 48*, 9 (Sept. 1999), 903–916.

[25] Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (Apr. 2010).

[26] Lauderdale, C., and Khan, R. Towards a codelet-based runtime for exascale computing. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT 2012)* (Mar. 2012), pp. 21–26.

[27] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1998)* (Oct. 1998), pp. 46–57.

[28] Mogul, J. C., Baumann, A., Roscoe, T., and Soares, L. Mind the gap: reconnecting architecture and os research. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS 2011)* (May 2011).

[29] Noakes, M. D., Wallach, D. A., and Dally, W. J. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 1993)* (May 1993), pp. 224–235.

[30] Regnier, P., Lima, G., and Barreto, L. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Operating Systems Review 42*, 6 (Oct. 2008), 52–63.

[31] Scheler, F., Hofer, W., Oechslein, B., Pfister, R., Shröder-Preikschat, W., and Lohmann, D. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2009)* (Dec. 2009), pp. 167–174.

[32] Swaine, J., Fetscher, B., St-Amour, V., Findler, R. B., and Flatt, M. Seeing the futures: Profiling shared-memory parallel Racket. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC 2012)* (Sept. 2012).

[33] Swaine, J., Tew, K., Dinda, P., Findler, R., and Flatt, M. Back to the futures: Incremental parallelization of existing sequential runtime systems. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)* (October 2010).

[34] Tew, K., Swaine, J., Flatt, M., Findler, R., and Dinda, P. Places: Adding message passing parallelism to racket. In *Proceedings of the 2011 Dynamic Languages Symposium (DLS 2011)* (October 2011).

[35] Thekkath, C. A., and Levy, H. M. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994)* (Oct. 1994), pp. 110–119.

[36] Treichler, S., Bauer, M., and Aiken, A. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013)* (Oct. 2013), pp. 495–514.

[37] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA 1998)* (July 1998), pp. 430–440.

[38] Wawrzynek, J., Patterson, D., Oskin, M., Lu, S.-L., Kozyrakis, C., Hoe, J. C., Chiou, D., and Asanović, K. Ramp: Research accelerator for multiple processors. *IEEE Micro 27*, 2 (Mar. 2007), 46–57.

[39] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2014)* (June 2014), pp. 2:1–2:8.