2018 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

An Evaluation of Asynchronous Software Events on Modern Hardware

Kyle C. Hale
Department of Computer Science
Illinois Insitute of Technology
khale@cs.iit.edu

Peter A. Dinda
Department of EECS
Northwestern University
pdinda@northwestern.edu

***Abstract*—Runtimes and applications that rely heavily on asynchronous 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 considerable overheads for demanding, high-performance parallel runtimes and applications. Other overheads can arise from a mismatched event programming or system call interface. Whatever the case, the average latency and variance in latency of commonly used software mechanisms for event notifications is abysmal compared to the capabilities of the hardware, which can exhibit orders of magnitude lower latency.**

We leverage the flexibility and freedom of the previously proposed Hybrid Runtime (HRT) model to explore the construction of low-latency, asynchronous software events uninhibited by interfaces and execution models commonly imposed by general-purpose OSes. We propose several mechanisms in a system we call *Nemo* which employs kernel mode-only features to accelerate event notifications by up to 4,000 times and we provide a detailed evaluation of our implementation using extensive microbenchmarks. We carry out our evaluation both on a modern x64 server and the Intel Xeon Phi. Finally, we propose a small addition to existing interrupt controllers (APICs) that could push the limit of asynchronous events closer to the latency of the hardware cache coherence network.

I. Introduction

Many runtimes leverage event-based primitives as an out-of-band notification mechanism that can signal events 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 often provide a foundation for building low-level synchronization primitives like mutexes and wait queues. The correct operation of parallel programs written for the shared-memory model relies crucially on low-latency, microarchitectural event notifications traversing the CPU's cache coherence network. Our focus in this paper is on asynchronous software events, by which we mean events that a sending thread or handler can trigger without blocking or polling, and for which the receiving thread or handler can wait without polling.

Ultimately these events are just an instance of unidirectional, asynchronous communication, so one might expect little room for performance improvement. We find, however, that 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 one 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 ultimately hinders 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 nor 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 capabilities of the hardware may actually be a fundamental issue with the *structure* of interactions between the application/runtime and the OS kernel.

Ultimately, the use of a general-purpose kernel (such as Linux), will necessarily involve a trade-off between functionality and performance that can hinder the construction of efficient asynchronous event mechanisms. In such an environment, the latency of performance critical operations for a particular application can suffer due to kernel overheads introduced by layers of abstraction, multiprogramming facilities, fixed execution models, and other instances where kernel code cannot make any assumptions about the types of applications and runtimes it will support.

In high-performance environments where low-latency message delivery is critical, such overheads become even more detrimental. Furthermore, many applications use structured communication patterns that necessitate synchrony among nodes and among hardware threads within a node. 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 [9], [10], [17]. In response to these challenges and the perception among high-performance application developers that operating systems can often "get in the way", we have seen the emergence of lightweight kernels such as Catamount [24], Kitten [25] and mOS [40], and hardware assists like RDMA for high-speed interconnects like Infiniband [19]. 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 II, 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 approach those limits with software abstractions implemented in an environment uninhibited by an underlying kernel, namely an HRT environment.

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, but are much faster. Unlike IPIs, where a thread invokes an interrupt handler on a remote core, these abstractions allow a thread to wake 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 can approach the hardware limit imposed by IPIs for the average latency and variance of latency for asynchronous event notifications. Unicast notifications in Nemo enjoy 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. Finally, Nemo can deliver broadcast events with much higher synchrony, exhibiting nearly identical latency to all destinations.

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 stems from 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.

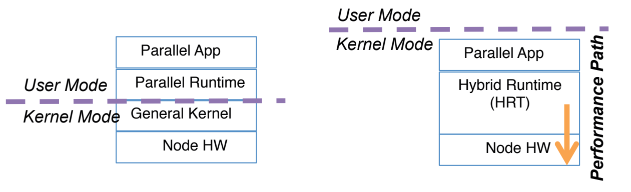
We make the following contributions:

- We outline the drawbacks of asynchronous event notifications in user-space, analyzing their performance and showing just how far they are from the capabilities of the hardware.
- We show how a runtime or application can alleviate some of these drawbacks by leveraging low-level hardware access and control (e.g. in an HRT).
- We present Nemo, an HRT-based event notification system which consists of three new notification mechanisms.
- 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 a kernel-mode environment, such as HRTs.

Nemo is currently available as an open-source extension of the Aerokernel platform, which we describe in the next section.

II. HRTs and Aerokernel

The hybrid runtime (HRT) model [15], [16] and the design and implementation of Aerokernel, an open-



(a) Current Model (b) Hybrid Runtime Model

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

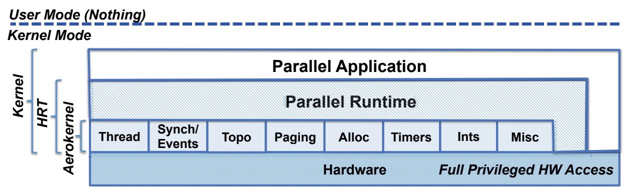


Fig. 2: Structure of Aerokernel.

source framework to support the model, was driven by the study of parallel runtimes including Legion [1], the NESL VCODE engine [4], the SWARM data flow runtime [26], ParalleX [21], Charm++ [22], the futures and places parallelism extensions to the Racket runtime [34], [35], [33], and nested data parallelism in Manticore [13], [12] and Haskell [5], [6].

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 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 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. Figure 1 depicts this model.

Aerokernel was developed to facilitate porting existing runtimes to become HRTs, and to develop new HRTs from scratch. Aerokernel implements basic kernel functionality and building blocks that can be leveraged by HRT developers. In the style of libOS [8], [20], 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 of our own design. Aerokernel links 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 can instead be delegated to a general-purpose OS running alongside the HRT. Figure 2 illustrates Aerokernel in the HRT context. Because Aerokernel provides a simple, fast, and easily enhanced kernel-mode environment, it gives us an ideal starting point for exploring the limits of event notifications.

III. Limits of Event Notifications

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 modern hardware already includes mechanisms for low-latency event communication. However, these hardware capabilities are traditionally reserved for kernel-only use. We discuss common uses 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 a kernel-mode environment 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. We expect that our findings could also apply to asynchronous events in other runtime environments that expose privileged hardware features or forego traditional privilege separation such as Dune [2], IX [3], and Arrakis [30].

A. 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 form groups of 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 ticks 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.

B. Runtime Events

In one common usage pattern of asynchronous event notifications, a signaling thread notifies one or more waiting (and not polling) 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 Charm++ [22], SWARM [26], and Legion [1], [37], all examples of modern parallel runtimes. 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 remote workers of available work or tasks that should execute.

Legion provides a good example. It uses an execution 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 of tasks ready to execute, the signaling 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 process bears some similarity to 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 show in Section III-C. Linux's futex abstraction attempts to ameliorate this cost with mixed success.

C. Microbenchmarks

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 wakes up. We map threads to distinct cores. The numbers represent statistics computed over 100 trials for each remote core (6,300 trials on *x64*, 22,700 on *phi*).

We compare three mechanisms. 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 builds on top of futexes. 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 *phi*, but only a small increment more on *x64*.

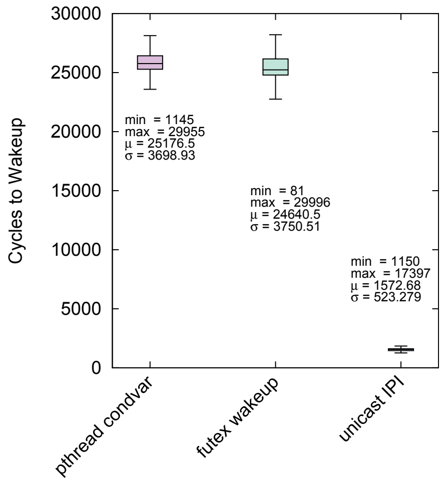
The third mechanism, denoted with "unicast IPI" on the figure, shows the unicast latency of an inter-processor interrupt (IPI). On *x64* and *phi*, each hardware thread has an associated interrupt controller (an APIC). The APIC commonly *receives* external interrupts and initiates their delivery to the 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 are typically used only by the kernel.

We also considered events triggered using the MONITOR/MWAIT pair of instructions present on modern AMD and Intel chips. These instructions allow a hardware thread to wait on a write to a particular range of memory, potentially entering a low-energy sleep state while waiting. Because the entire hardware thread is essentially blocked when executing the MWAIT instruction, we did not include a comparison with this technique.

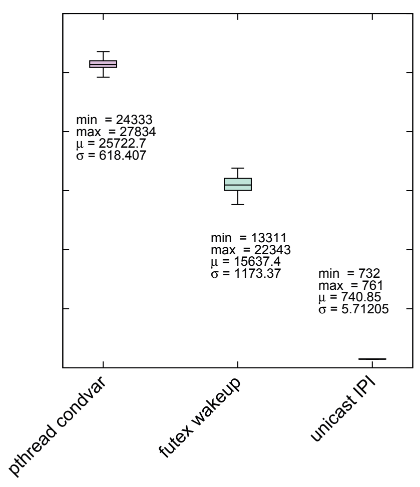
Figure 3 shows that IPIs are, on average, much faster than either condition variables or futexes. On *x64*, they have roughly 16 times lower latency than either, while on *phi*, they have roughly 32 times lower latency than condition variables and 16 times lower latency than futexes. On *phi*, the average IPI latency is only 740 cycles. The wall-clock time on the two machines is similar, as *x64* has roughly twice the clock rate.

IPIs are *not* doing the same thing as a condition variable or a futex. The closest user-level analog to the IPI is signal delivery. While the runtimes we examined do not directly use signal delivery, it is instructive to compare. The minimum measured one-way signal delivery latency was 10.8K cycles on *x64* and 10.3K cycles on *phi*. These latencies are an order of magnitude higher than the IPI latency these platforms are capable of. 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 measure the IPI time because this latency 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 ($\sim$20x). We will attempt to achieve this improvement in Section IV by moving towards a *purely* asynchronous mechanism enabled by the HRT environment.

We observe that not only is the average time much lower for an IPI, but its variance is also diminished



(a) x64



(b) phi

Fig. 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.

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 fewer barriers to predictable performance.

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*, again comparing a condition variable based approach in pthreads, Linux futex, and IPIs. Measurements here operate as with the unicast events, but we additionally keep track of time of delivery on every destination so we can assess the synchronicity of the broadcasts.

The relative latency improvements for broadcasts with condition variables and futexes are similar to the unicast case, but the gain from using broadcast IPIs is much larger. On *phi* (not shown), the average latency of a broadcast IPI received by all targets is over 4,000 times lower than for a mechanism based on condition variables. The gain in variance 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 choose a barrier over a condition variable to signal a wakeup on multiple cores, barriers lack the asynchrony needed for these kinds of event notifications.

We should also hope that a broadcast event causes

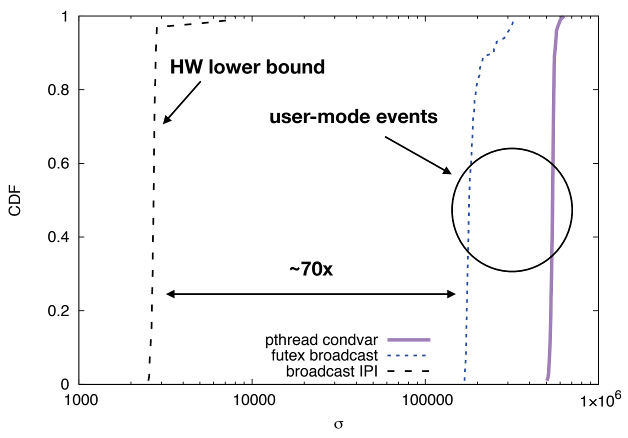
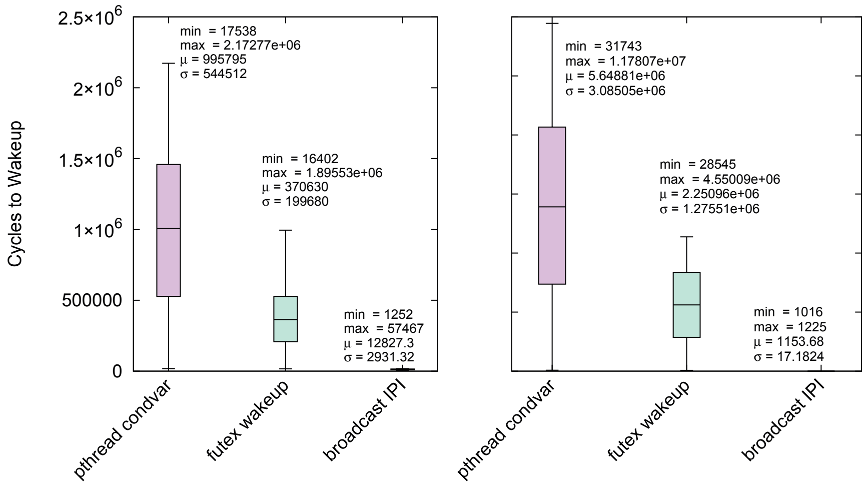


Fig. 5: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *x64*. Broadcast IPIs achieve a synchrony that is three orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

wakeups 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 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,



(a) x64 (b) phi

Fig. 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 latency than futexes. Variance in latency is similarly reduced.

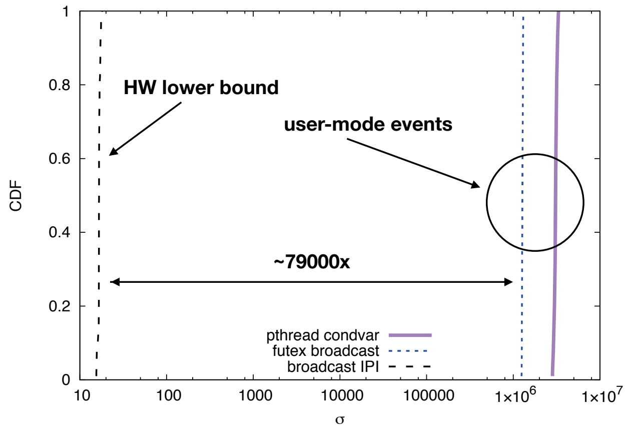


Fig. 6: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *phi*. Broadcast IPIs achieve a synchrony that is five orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

there are 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.

D. Discussion

The large 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 timescale of a slow network packet delivery, but also that the programmer can do little to ensure that these wakeups occur with *predictable* performance. The problem is 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 asynchronous 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.

IV. Nemo Events

Nemo is an asynchronous event notification system for HRTs built within an Aerokernel framework. Nemo addresses the performance issues 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 IPI network on this machine, we are confident that these inflection points

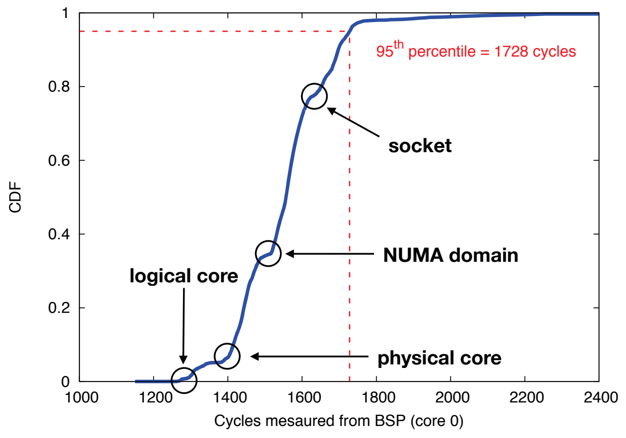


Fig. 7: CDF of unicast IPI latency from the Bootstrap Processor (BSP) to all other cores on *x64*. Approaching this profile is the goal of Nemo.

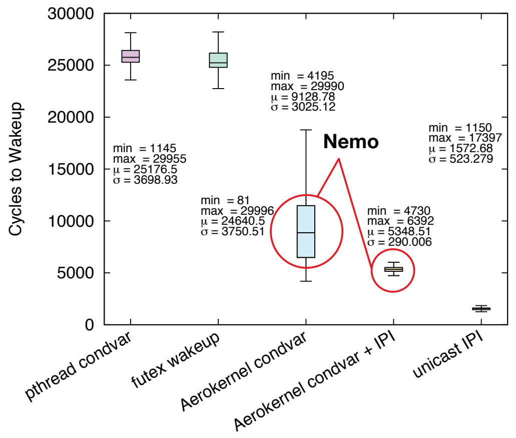


Fig. 8: Nemo kernel-mode event mechanisms for single wakeups on *x64*. Average latency is reduced by over a factor of four, and variation is considerably reduced.

correspond to distances within the chip hierarchy as indicated in the captions. As Nemo begins to exhibit similar behavior, we will know we 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 stems from its use of a single-chip processor with a balanced interconnect joining the cores.

A. *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 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 signaled 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 variable 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.

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 and other implementation overheads. The implementation does, however, exhibit considerable variance in wakeup latency. This is because the wakeup time depends on how long it takes for the CPU's background thread to `yield()` again.

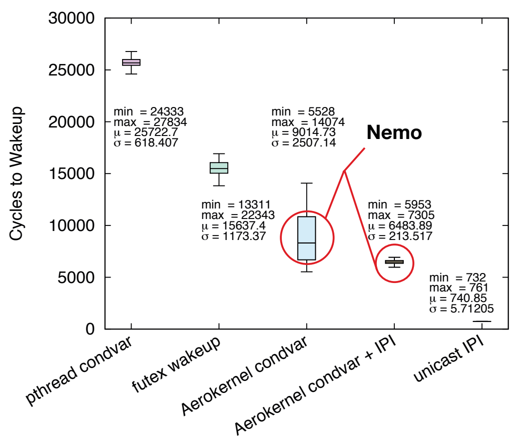


Fig. 9: Nemo kernel-mode event mechanisms for single wakeups on *phi*. Average latency is reduced by over a factor of four and variation is considerably reduced.

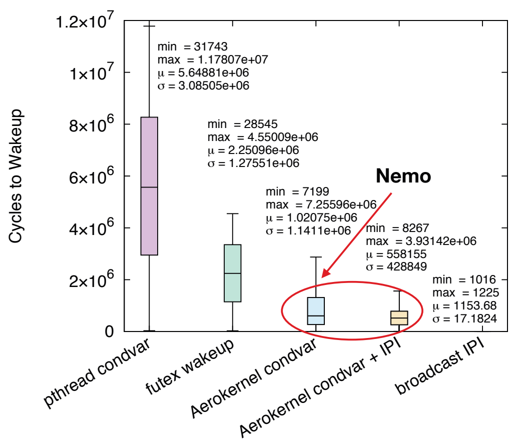


Fig. 11: Nemo kernel-mode event mechanisms for broadcast wakeups on *phi*. Average latency is reduced by an factor of 16 and variation is considerably reduced.

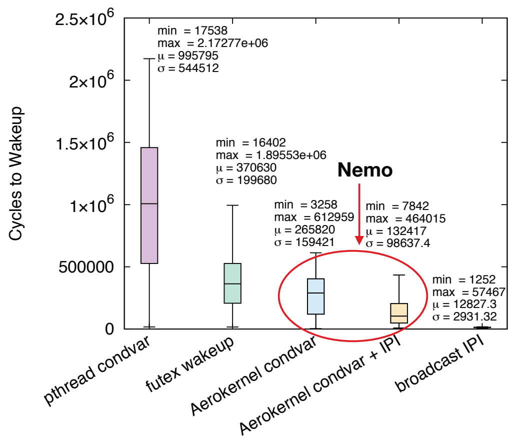


Fig. 10: Nemo kernel-mode event mechanisms for broadcast wakeups on *x64*. Average latency is reduced by a factor of 10 and variation is considerably reduced.

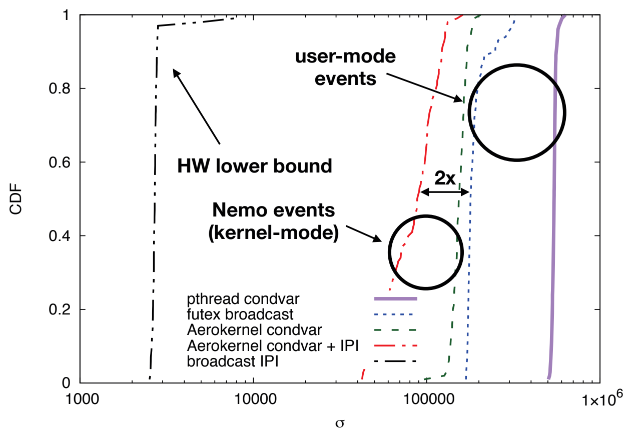


Fig. 12: CDF showing the Nemo kernel-mode event mechanisms for broadcast wakeups on *x64*. Nemo achieves an order of magnitude better synchrony in thread wakeups.

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 III-C 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 indicated in the figures and previous discussions.

B. 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

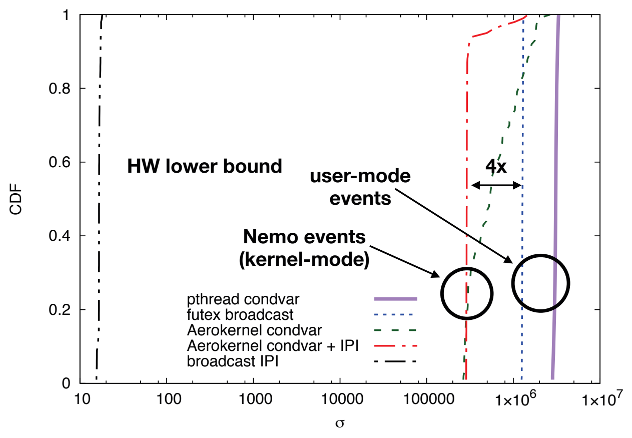


Fig. 13: CDF showing the Nemo kernel-mode event mechanisms for broadcast wakeups on *phi*.

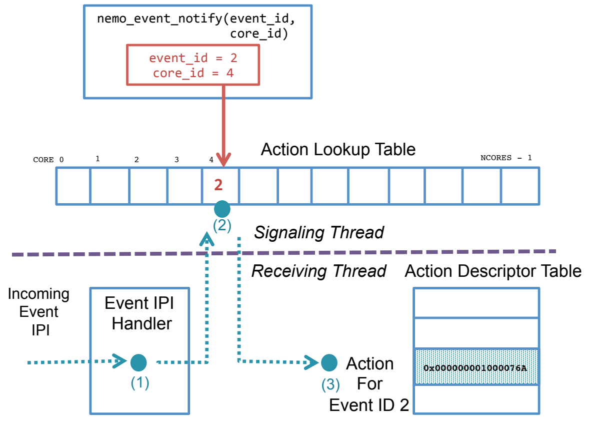


Fig. 14: Nemo Active Message-inspired event wakeups implemented using IPIs.

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 built directly on top of IPIs. The mechanism is also informed by how 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. For space reasons, we omit performance figures for *phi* in this section, as the shapes of the graphs for both machines are similar.

We claim that Active Messages [38] would better match the functional behavior that event-based runtimes need. Active Messages enable 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. Furthermore, since the HRT is not a multi-programmed environment, we can be sure that the receiving thread is a participant in the parallel runtime/application, and thus has the high-level information necessary to process the event. We can eliminate handling overhead by leveraging existing logic in the hardware already meant for handling asynchronous events—in this case, IPIs. IPIs by themselves, however, cannot implement a complete Active Message substrate, as there is no payload other than the interrupt vector and state pushed on the stack by hardware.

Figure 14 shows the design and control flow of our Active Message-inspired event mechanism. We reserve a slot in the Interrupt Descriptor Table (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.

Figure 15 shows a CDF of the latency of Nemo's Active Message-inspired events compared to the unicast IPI. In all cases Nemo events are only roughly 40 cycles slower on *phi* (not shown) 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). Note that the user-space signal delivery latencies discussed in Section III-C are more than double the latency of Nemo's "Aerokernel condvar + IPI" mechanism. Furthermore, Nemo's Active Message-inspired event wakeups approach IPI latency, and thus are also an order of magnitude faster than user-space signals.

Figure 16 shows the latency of Active Message-

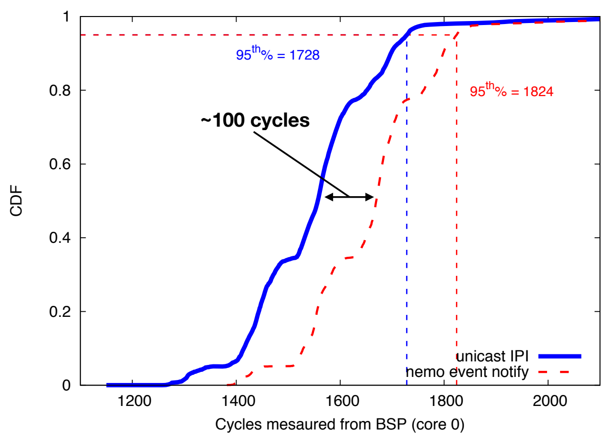


Fig. 15: CDF showing unicast latency of Nemo's Active Message-inspired events compared to unicast IPIs on *x64*. Nemo lies within ~5% of IPI performance.

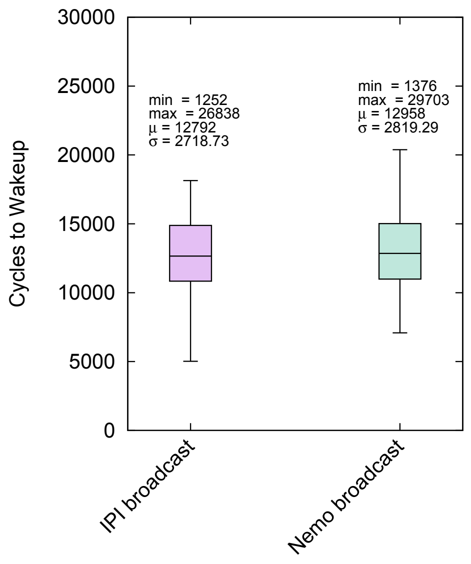


Fig. 16: Broadcast latency of Nemo's Active Message-inspired events compared to broadcast IPIs on *x64*. Performance is nearly identical.

inspired Nemo events compared to broadcast IPIs. The performance of Nemo events are within tens of cycles of broadcast IPIs, as we would expect. Figure 17 shows the amount of synchrony present in Nemo events. We give an explanation of this type of figure in Section III-C.

V. 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 programs, we would like to calibrate this performance to another hardware capability that is critical to the performance of multicore machines, namely the cache coherence network. The question we ask here is can we improve the hardware limit?

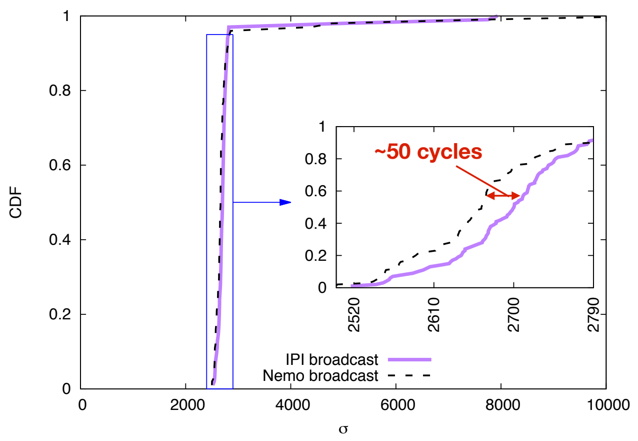


Fig. 17: CDF comparing Nemo's Active Message-inspired broadcast events to broadcast IPIs on x64. Synchrony is nearly identical.

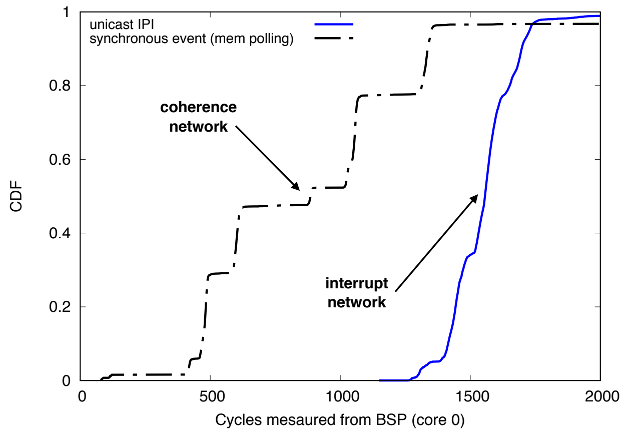


Fig. 18: CDF comparing latency of asynchronous unicast IPIs compared to a simple synchronous notification scheme using memory polling on x64. This represents the basic cost difference between a synchronous and asynchronous event imposed by the hardware.

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 have low latency but also predictable 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

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

Fig. 19: Estimated IPI cost breakdown in cycles on *x64*.

memory location waiting for its value to change. When a signaling thread changes this value, its core's cache controller 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 18 shows the performance of this synchronous mechanism compared to the asynchronous mechanism of unicast IPIs on our x64 hardware. IPIs are roughly 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 prevents 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 19 shows the results.

The latency of a unicast IPI involves three components. The first, "Source APIC write", is the time to initiate the IPI 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 "Communication delay" is simply these two numbers subtracted from the total unicast IPI cost shown in Section III-C. It is likely to be an overestimate.

Integrating the observations of Figures 18 and 19 suggests that the reason why an asynchronous IPI has so much higher latency than a synchronous coherence event is likely to be due, in large part, 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

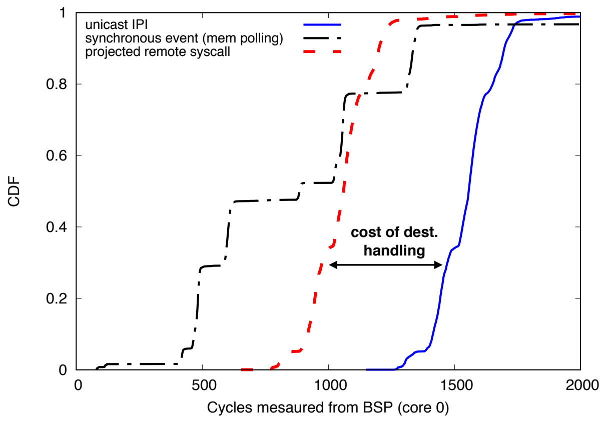


Fig. 20: CDF of the projected latency of the proposed "remote syscall" mechanism. This comparison is made relative to the measured latencies of Figure 18.

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 unnecessary.

A similar issue 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 19 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 a similar modification to the architecture could produce comparable benefits for low-latency event delivery and handling. 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. Figure 20 shows the projected improvements. 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 to deliver. Mode `011` is currently reserved, so this is a possible candidate for a *remote* `syscall` mode. There are, of course 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 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.

VI. Related Work

The real-time OS community has studied asynchronous events in depth, focusing on events in contexts such as predictable interrupt handling [31], priority inversion [32], and fast vectoring to user-level code [14]. However, this work does not consider the design of asynchronous events in a context where the application/runtime has kernel-mode privileges and full access to hardware, as in an HRT.

Thekkath and Levy introduced a mechanism [36] 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 I. The motivation for this technique mirrors motivations for RDMA-based techniques we see used in practice today. In contrast, Nemo's design focuses on asynchronous events.

Horowitz introduced a programmable hardware technique for *Informing Memory Operations*, which vector to a user-level handler with minimal latency on cache miss events [18]. These events bear more similarities to exceptions than to asynchronous events, especially those originating at a remote core.

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 [23]. Chaterjee discusses further details of this technique, particularly as it applies to MIT's J-Machine [29] and M-Machine [11]. A modern example of this technique can be found in the MONITOR and MWAIT instruction pair. We discuss how this type of technique differs from our goals in Section III-C.

The Message Driven Processor (MDP), from which the J-Machine was built, had hardware contexts specifically devoted to handling message arrivals [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 V 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 [39]—have not made their way into commodity processors used in today's supercomputers, servers, and accelerators.

VII. Conclusions and future work

We have shown how the performance of asynchronous software events suffers when the application/runtime is restricted to user-space mechanisms and mismatched event programming interfaces. The performance of these mechanisms comes nowhere near the hardware limits of IPIs, much less cache coherence messages. By leveraging the HRT model, wherein the runtime and application can execute with fully privileged hardware access, we increased the performance of these event mechanisms considerably. We did so by designing, implementing, and evaluating the Nemo asynchronous event system within the 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, allowing a greater degree of synchrony between broadcasts to multiple cores. In addition to Nemo, we also 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. We showed that such additions can push the performance of asynchronous software events closer to that of the hardware cache coherence network.

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

References

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

[2] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)*, pages 335–348, Oct. 2012.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 49–65, Oct. 2014.

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

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

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

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

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

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

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

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

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

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

[14] G. Fry and R. West. 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.

[15] K. C. Hale and P. A. Dinda. A case for transforming parallel runtimes into operating system kernels. In *Proceedings of the 24th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)*, June 2015.

[16] K. C. Hale and P. A. Dinda. 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)*, pages 161–175, Apr. 2016.

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

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

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

[20] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 52–65, Oct. 1997.

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

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

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

[24] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting (CUG 2005)*, May 2005.

[25] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. 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] C. Lauderdale and R. Khan. 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)*, pages 21–26, Mar. 2012.

[27] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. 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)*, pages 46–57, Oct. 1998.

[28] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares. 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] M. D. Noakes, D. A. Wallach, and W. J. Dally. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA 1993)*, pages 224–235, May 1993.

[30] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 1–16, Oct. 2014.

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

[32] F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Shröder-Preikschat, and D. Lohmann. 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), pages 167–174, Dec. 2009.

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

[34] J. Swaine, K. Tew, P. Dinda, R. Findler, and M. Flatt. 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.

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

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

[37] S. Treichler, M. Bauer, and A. Aiken. 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)*, pages 495–514, Oct. 2013.

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

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

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