Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support

Kyle C. Hale Peter A. Dinda

Department of Electrical Engineering and Computer Science
Northwestern University
{k-hale, pdinda}@northwestern.edu

Abstract

In our hybrid runtime (HRT) model, a parallel runtime system and the application are together transformed into a specialized OS kernel that operates entirely in kernel mode and can thus implement exactly its desired abstractions on top of fully privileged hardware access. We describe the design and implementation of two new tools that support the HRT model. The first, the Nautilus Aerokernel, is a kernel framework specifically designed to enable HRTs for x64 and Xeon Phi hardware. Aerokernel primitives are specialized for HRT creation and thus can operate much faster, up to two orders of magnitude faster, than related primitives in Linux. Aerokernel primitives also exhibit much lower variance in their performance, an important consideration for some forms of parallelism. We have realized several prototype HRTs, including one based on the Legion runtime, and we provide application macrobenchmark numbers for our Legion HRT. The second tool, the hybrid virtual machine (HVM), is an extension to the Palacios virtual machine monitor that allows a single virtual machine to simultaneously support a traditional OS and software stack alongside an HRT with specialized hardware access. The HRT can be booted in a time comparable to a Linux user process startup, and functions in the HRT, which operate over the user process's memory, can be invoked by the process with latencies not much higher than those of a function call.

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. We also thank Madhav Suresh and Conor Hetland for their help with NESL and NDPC.

VEE '16, April 02 - 03, 2016, Atlanta, GA, USA

ACM 978-1-4503-3947-6/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2892242.2892255

1. Introduction

Considerable innovation in parallelism is occurring today, targeting a wide range of scales from mobile devices to exascale computing. How to execute parallel languages with high performance and efficiency is a question of wide interest. Our focus is on parallel runtime systems, the medium through which these languages interact with the operating system and the hardware. Many interaction models are possible, and the innovation and change driven by parallelism itself makes feasible the adoption of other models. We are studying one such model in depth.

A hybrid runtime (HRT) is a parallel runtime system, along with its application, that runs entirely in *kernel mode* on the target hardware. That is, an HRT is a parallel runtime stack that has either been developed as an operating system kernel or has been ported to become an OS kernel—it is a hybrid of a kernel and a parallel runtime. Because an HRT has fully privileged access to the machine, it can use all hardware features available on the machine, it can create whatever OS level abstractions are suitable to it, and it can use those abstractions without any system call overheads. In contrast, in today's common model of a parallel runtime running in user mode on top of a general purpose or even lightweight kernel, the parallel runtime cannot use privileged features of the hardware, it is limited to the abstractions exposed through the system call interface, and even those abstractions come at the cost of a system call.

We previously argued the *case* for HRTs [37], and we now describe the design, implementation, and evaluation of two *tools* we have developed to support the creation and execution of HRTs, and evaluate the HRT model. The first tool, the Nautilus Aerokernel (we usually just write "Aerokernel"), is a kernel framework specifically designed to support the creation of HRTs. It provides a basic kernel that can be booted within milliseconds after boot loader execution on a multicore, multisocket machine, accelerator, or virtual machine. Aerokernel includes basic building blocks such as simple memory management, threads, synchronization, IPIs and other in-kernel abstractions that a parallel runtime can be ported to or be built on top of to become an HRT. While

Aerokernel provides functionality, it does not require the HRT to use it, nor does it proscribe the implementation of other functionality. Aerokernel was developed for 64-bit x86 machines (x64) and then ported to the Intel Xeon Phi.

Our evaluation of Aerokernel includes three aspects. First, we give detailed microbenchmark evaluations, comparing its functionality and performance to that of analogous facilities available at user-level on Linux that are typically used within parallel runtimes. Aerokernel functionality such as thread creation and events operate up to two orders of magnitude faster than Linux due to their implementation and by virtue of the fact that there is no kernel/user boundary to cross. They also operate with much less variation in performance, an important consideration for many models of parallelism, particularly with scale. The second aspect of our evaluation is to consider the challenges of porting parallel runtimes to Aerokernel. We describe ports of three runtimes, the Legion runtime [4], the NESL VCODE engine [12], and a home-grown nested data parallel language. Even the most complex of these, Legion, was feasible to port to Aerokernel to become an HRT. Finally, we do application benchmarking using Sandia National Lab's HPCG, comparing Legion on Linux and the Legion/Aerokernel HRT on x64 and Xeon Phi.

While running an HRT on bare metal is suitable for some contexts (e.g., an accelerator or a node of a supercomputer), we may also want to use an HRT in shared contexts or ease the porting of runtimes that have significant dependencies on an existing kernel. Our second tool, the hybrid virtual machine (HVM), facilitates these use cases. HVM is an extension to the open source Palacios virtual machine monitor (VMM) [48] that makes it possible to create a VM that is internally partitioned between virtual cores that run a "regular" operating system (ROS), and virtual cores that run an HRT. The ROS cores see a subset of the VM's memory and other hardware, while HRT cores see all of it and may be granted specialized physical hardware access by the VMM. The ROS application can invoke functions in the HRT that operate over data in the ROS application. Finally, the HRT cores can be booted independently of the ROS cores using a model that allows an HRT to begin executing in 10s of microseconds, and with which an Aerokernel-based HRT can be brought up in 10s of milliseconds. This makes HRT startup in the HVM comparable in cost to `fork()/exec()` functionality in the ROS. In effect, the HVM allows a portion of an x64 machine to act as an accelerator.

Our contributions are as follows.

- We describe in detail the design of the Nautilus Aerokernel kernel framework and how it differs from other kernels.
- We describe the implementation of Aerokernel on x64 and Phi. As far as we are aware, Aerokernel is only the second non-Intel kernel on the Phi.

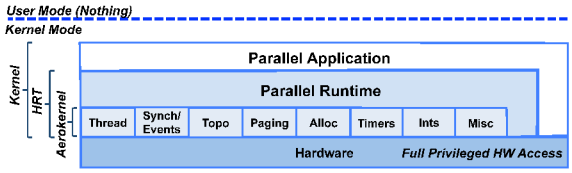


Figure 1: Structure of Aerokernel.

- We microbenchmark Aerokernel on x64 and Phi, particularly focusing on threads and events.
- We describe our experiences with porting three runtimes, including Legion, to become Aerokernel-based HRTs.
- We give application benchmark results for HPCG on Legion, comparing Aerokernel with Linux, on x64 and Phi.
- We describe the design, implementation, and performance evaluation of the HVM concept.

The Nautilus Aerokernel and the HVM extensions to Palacios are open-source and publicly available at `v3vee.org`.

2. Nautilus Aerokernel

Aerokernel design and implementation has been driven by studying parallel runtimes, including the three (Legion, NESL, NPDC) described in Section 4, the SWARM data flow runtime [49], ParalleX [43], Charm++ [44], the futures and places parallelism extensions to the Racket runtime [64, 65, 63], and nested data parallelism in Manticore [34, 33] and Haskell [20, 21]. We have studied these codebases and in the case of Legion, NPDC, SWARM, and Racket, we also interviewed their developers to understand their views of the limitations of existing kernel support.

Aerokernel is *not* a general purpose kernel. In fact, there is not even a user space. Instead, its design focuses specifically on helping parallel runtimes to achieve high performance as HRTs. The non-critical path functionality of the runtime is assumed to be delegated, for example to the host in the case of an accelerator or to the ROS portion of an HVM. The abstractions Aerokernel provides are based on our analysis of the needs of the runtimes we examined. Our abstractions are optional. Because an HRT runs entirely at kernel level, the developer can also directly leverage all hardware functionality to create new abstractions.

Our choice of abstractions was driven in part to make it feasible to port existing parallel runtimes to become Aerokernel-based HRTs. A more open-ended motivator was to facilitate the design and implementation of new parallel runtimes that do not have a built-in assumption of being user space processes.

2.1 Design

Aerokernel is designed to boot the machine, discover its capabilities, devices, and topology, and immediately hand control over to the runtime. Figure 1 shows the basic struc-

ture, showing the functionality provided by Aerokernel in the context of the runtime and application. Note that Aerokernel is a thin layer in the HRT model, and that in this model there is no user space. The runtime and the application have full access to hardware and can pick and choose which Aerokernel functionality to use. The entire assemblage of the figure is compiled into a multiboot2-compliant kernel.

We focus the following discussion on functionality where Aerokernel differs most from other kernels. In general, the defaults for Aerokernel functionality strive to be simple and easy to reason about from the HRT developer's viewpoint.

Threads In designing a threading model for Aerokernel, we considered the experiences of others, including work on high-performance user-level threading techniques like scheduler activations [2] and Qthreads [67]. Ultimately, we designed our threads to be lightweight in order to provide an efficient starting point for HRTs. Aerokernel threads are *kernel* threads. A context switch between Aerokernel threads *never* involves a change of address spaces. Aerokernel threads can be configured to operate either preemptively or cooperatively, the latter allowing for the elimination of timer interrupts and scheduling of threads exactly as determined by the runtime.

The nature of the threads in Aerokernel is determined by how the runtime uses them. This means that we can directly map the logical view of the machine from a runtime's point of view (see Section 4.1) to the physical machine. This is not typically possible to do with any kind of guarantees when running in userspace. In fact, this is one of the concerns that the Legion runtime developers expressed with running Legion on Linux. The default scheduler and mapper binds a thread to a specific hardware thread as selected by thread creator, and schedules round-robin. The runtime developer can easily change these policies.

Another distinctive aspect of Aerokernel threads is that a thread fork (and join) mechanism is provided in addition to the common interface of starting a new thread with a clean new stack in a function. A forked thread has a limited lifetime and will terminate when it returns from the current function. It is incumbent upon the runtime to manage the parent and child stacks correctly. This capability is leveraged in our ports of NESL and NDPC.

Thread creation, context switching, and wakeup are designed to be fast and to leverage runtime knowledge. For example, maximum stack sizes and context beyond the GPRs can be selected at creation time. Because interrupt context uses the current thread's stack, it is even possible to create a single large-stacked idle thread per hardware thread and then drive computation entirely by inter-processor interrupts (IPIs), one possible mapping of an event-driven parallel runtime such as SWARM.

Synchronization and events Aerokernel provides several variants of low-level spinlocks, including MCS locks and bakery locks. These are similar to those available in other kernels, and comparable in performance.

Aerokernel focuses to a large extent on asynchronous events, which are a common abstraction that runtime systems often use to distribute work to execution units, or workers. For example, the Legion runtime makes heavy use of them to notify logical processors (Legion threads) when there are Legion tasks that are ready to be executed. Userspace events require costly user/kernel interactions, which we *eliminate* in Aerokernel.

Aerokernel provides two implementations of condition variables that are compatible with those in pthreads. These implementations are tightly coupled with the scheduler, eliminating unnecessary software interactions. When a condition is signaled, the default Aerokernel condition variable implementation will simply put the target thread on its respective hardware thread's ready queue. This, of course, is not possible from a user-mode thread.

When a thread is signaled in Aerokernel it will not run until the scheduler starts it. For preemptive threads, this means waiting until the next timer tick, or an explicit yield from the currently running thread. Our second implementation of condition variables mitigates this delay by having the signaling thread "kick" the appropriate core with an IPI after it has woken up the waiting thread. The scheduler recognizes this condition on returning from the interrupt and switches to the awakened thread.

The runtime can also make direct use of IPIs, giving it the ability to force immediate execution of a function of its choosing on a remote destination core. Note that the IPI mechanism is unavailable when running in user-space.

Topology and memory allocation Modern NUMA machines organize memory into separate domains according to physical distance from a physical CPU socket, core, or hardware thread. This results in variable latency when accessing memory in the different domains and also means achieving high memory bandwidth requires leveraging multiple domains simultaneously. Platform firmware typically enumerates these NUMA domains and exposes their sizes and topology to the operating system in a way that supports both modern and legacy OSes.

Aerokernel captures this topology information on boot and exposes it to the runtime. The page and heap allocators in Aerokernel allow the runtime to select which domains to allocate from, with the default being that allocations are satisfied from the domain closest to the current location of the thread requesting the allocation. All allocations are done immediately. This is in contrast to the policy of deferred allocations whose domains are determined on first touch, the typical default policy for general purpose kernels. A consequence is that a runtime that implements a specific execution policy, for example the owner-computes rule (e.g., as in HPF [39]) or inspector-executor [26], can more easily

reason about how to efficiently map a parallel operation to the memory hardware.

A thread's stack is allocated using identity-mapped addresses based on the initial binding of the thread to a hardware thread, again to the closest domain. Since threads do not by default migrate, stack accesses are low latency, even across a large stack. If the runtime is designed so that it does not allow or can fix pointers into the stack, even the stack can be moved to the most friendly domain if the runtime decides to move the thread to a different hardware thread.

We saw NUMA effects that would double the execution time of a long-running parallel application on the Legion runtime. While user-space processes do typically have access to NUMA information and policies, runtimes executing in the Aerokernel framework have *full* control over the placement of threads and memory and can thus enjoy guarantees about what can affect runtime performance.

Paging Aerokernel has a simple, yet high-performance paging model aimed at high-performance parallel applications. When the machine boots up, each hardware thread identity-maps the entire physical address space using large pages (2MB and 1 GB pages currently, 512 GB pages when available in hardware) to create a single *unified* address space. Optionally, the identity map can be offset into the "higher half" of the x64 address space (Section 5.3). An Aerokernel-based kernel can also be linked to load anywhere in the physical address space.

The static identity map eliminates expensive page faults and TLB shootdowns, and reduces TLB misses. These events not only reduce performance, but also introduce unpredictable OS noise [31] from the perspective of the runtime developer. OS noise is well known to introduce timing variance that becomes a serious obstacle in large-scale distributed machines running parallel applications. The same will hold true for single nodes as core counts continue to scale up. The introduction of variance by OS noise (not just by asynchronous paging events) not only limits the performance and predictability of existing runtimes, but also limits the *kinds* of runtimes that can take advantage of the machine. For example, runtimes that need tasks to execute in synchrony (e.g., in order to support a bulk-synchronous parallel [35] application or a runtime that uses an abstract vector model) will experience serious degradation if OS noise comes into play.

The use of a single unified address space also allows fast communication between threads, and eliminates much of the overhead of context switches. The only context switches are between kernel threads, so no page table switch or kernel-triggered TLB flush ever occurs. This is especially useful when Aerokernel runs virtualized, as a large portion of VM exits come from paging related faults and dynamic mappings initiated by the OS, particularly using shadow paging. A shadow-paged Aerokernel exhibits the minimum possible shadow page faults, and shadow paging can be more efficient that nested paging, except when shadow page faults are common [3].

Timers Aerokernel optionally enables a per-hardware thread scheduler tick mechanism based on the Advanced Programmable Interrupt Controller (APIC) timer. This is only needed when preemption is configured.

For high resolution time measurement across hardware threads, Aerokernel provides a driver for the high-precision event timer (HPET) available on most modern x64 machines. This is a good mapping for real-time measurement in the runtimes we examined. Within per-hardware thread timing, the cycle counter is typically used.

Interrupts External interrupts in Aerokernel work just like any other operating system, with the exception that by default only the APIC timer interrupt is enabled at bootup (and only when preemption is configured). The runtime has complete control over interrupts, including their mapping, assignment, and priority ordering.

2.2 Implementation

The process of building Aerokernel as a minimal kernel layer with support for modern x64 NUMA machines took six person-months of effort on the part of seasoned OS/VMM kernel developers. Aerokernel, which was developed from scratch, comprises about 25,000 lines of code: about 23,000 lines of C, 1000 lines of assembly, 200 lines of C++, and the rest in various scripting languages. Building a kernel, however, was not our main goal. Our main focus was supporting the porting and construction of runtimes for the HRT model.

The Legion runtime was the most challenging and complex of the three runtimes to bring up in Aerokernel. Legion is almost twice the size of Aerokernel, consisting of about 43,000 lines of C++. Porting Legion and the other runtimes took a total of about four person-months of effort—three person-months as described in Section 4, and one person-month in extensions to Aerokernel. In the end a modest 800 lines of additional code (650 C, 150 C++) needed to be added to Aerokernel, primarily to support C++.

This suggests that exploring the HRT model for existing or new parallel runtimes, especially with a small kernel like Aerokernel designed with this in mind, is a perfectly manageable task for an experienced systems developer.

2.3 Xeon Phi

We have ported Aerokernel to the Intel Xeon Phi. Although the Phi is technically an x64 machine, it has differences that make porting a kernel to it challenging. These include the lack of much PC legacy hardware, a distinctive APIC addressing model, a distinctive frequency/power/thermal model, and a bootstrap and I/O model that is closely tied to Intel's MPSS stack. Our port consists of two elements.

Philix is a set of tools to support booting and communicating with a third-party kernel on the Phi in compliance with Intel's stack, while at the same time not requiring the

kernel to itself support the full functionality demanded of MPSS. Philix also includes basic driver support for the Phi that can be incorporated into the third-party kernel. This includes console support on both the host and Phi sides to make debugging a new Phi kernel easier. Philix comprises 1150 lines of C.

Our changes to add Phi support to Aerokernel comprised about 1350 lines of C. This required about 1.5 person months of kernel developer effort, mostly spent in ferreting out the idiosyncrasies of the Phi.

3. Microbenchmarks

We now evaluate the performance of the basic primitives in Aerokernel that are particularly salient to HRT creation, comparing them to Linux user-level and kernel-level primitives. The performance of basic primitives is important because runtimes build on these mechanisms. Although they can use the mechanisms cleverly (Legion's task model is effectively a thread pool model, for example), making the underlying primitives and environment faster can make runtimes faster, as we shall see.

Experimental setup We measure performance on an x64 NUMA machine and on an Intel Xeon Phi. The x64 configuration is a 2.1GHz AMD Opteron 6272 (Interlagos) server machine with 64 cores and 128 GB of memory. The cores are spread across 4 sockets, and each socket comprises two NUMA domains. All CPUs within one of these NUMA domains share an L3 cache. Within the domain, CPUs are organized into 4 groups of 2 hardware threads. The hardware threads share an L1 instruction cache and a unified L2 cache. Hardware threads have their own L1 data cache. We configured the BIOS for this machine to "Maximum performance" to eliminate the effects of power management. This machine also has a "freerunning' TSC, which means that the TSC will tick at a constant rate regardless of the operating frequency of the processor core. For Linux tests, it runs Red Hat 6.5 (1.5 years old at the time of this writing) with the stock Linux kernel binary version 2.6.32. It is important to note that this kernel has been highly optimized by Red Hat. For example, it uses the transparent huge page mechanism.

For the Xeon Phi tests, we use a Xeon Phi 3120A PCI accelerator along with the Intel MPSS 3.4.2 toolchain, which uses a modified 2.6.38 Linux kernel. It is important to point out that this is the current kernel binary shipped by Intel for use with Intel Xeon Phi hardware.

We use the `rdtscp` instruction to enforce proper serialization of instructions when timing using the cycle counter. Measurements are taken over at least 1000 runs with results shown as box plots or CDFs.

Threads Figure 2 compares thread creation latency between Linux userspace (pthreads), Linux kernel threads, and Aerokernel threads. We compare with pthreads because runtimes (such as Legion) build on these mechanisms. While

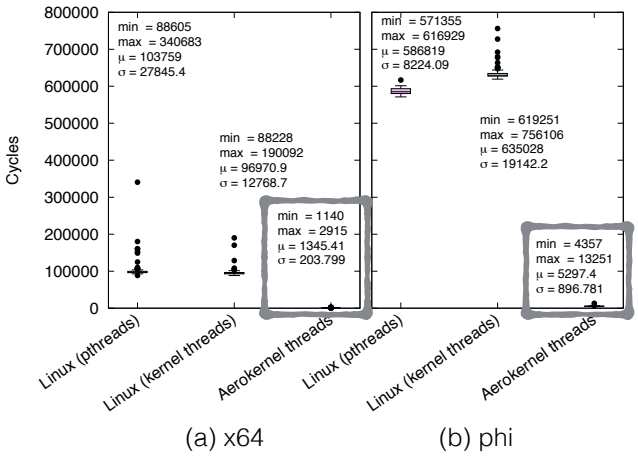
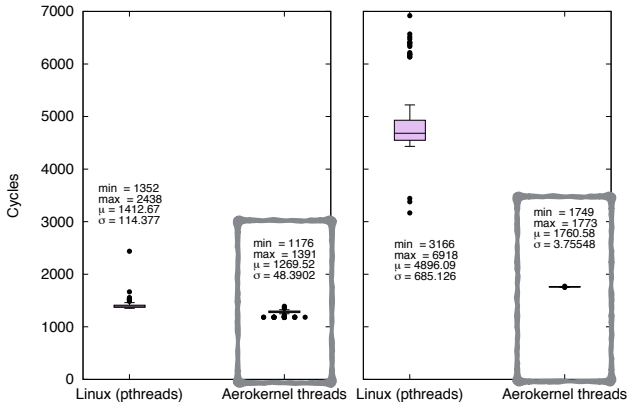


Figure 2: Thread creation latency. Aerokernel thread creations are on average two orders of magnitude faster that Linux userspace (pthreads) or kernel thread creations and have at least an order of magnitude lower variance.

thread creation may or may not be on the performance critical path for a particular runtime the comparison demonstrates the lightweight capabilities of Aerokernel. The cost measured is the time for the thread creation function to return to the creator thread. For Aerokernel, this includes placing the new thread in the run queue of its hardware thread. A thread fork in Aerokernel has similar latency since the primary difference compared to ordinary thread creation has to do the content of the initial stack for the new thread. The time for the new thread to begin executing is bounded by the context switch time, which we measure below.

On both platforms, thread creation in Aerokernel has two orders of magnitude lower latency on average than both Linux options, and, equally important, the latency has little variance. Thread creation in Aerokernel also scales well, as, like the others, it involves constant work. From an HRT developer's point of view, these performance characteristics potentially makes the creation of smaller units of work feasible, allows for tighter synchronization of their execution, and allows for large numbers of threads.

Figure 3 illustrates the latencies of context switches between threads on the two platforms, comparing Linux and Aerokernel. In both cases, no floating point or vector state is involved—the cost of handling such state is identical across Linux and Aerokernel. The average cost of an Aerokernel context switch on the x64 is about 10% lower than that of Linux, but Aerokernel exhibits a variance that's lower by a factor of two. On the Phi, Aerokernel exhibits two orders of magnitude lower variance in latency and more than factor of two lower average latency. The instruction count for a thread context switch in Aerokernel is much lower than that for Linux. On the x64, this does not have much effect because the hardware thread is superscalar. On the other hand, the hardware thread on the Phi is not only not su-



(a) x64 (b) phi

Figure 3: Thread context switch latency. Aerokernel thread context switches similar in average performance to Linux on x64 and over two times faster on Phi. In both cases, the variance is considerably lower.

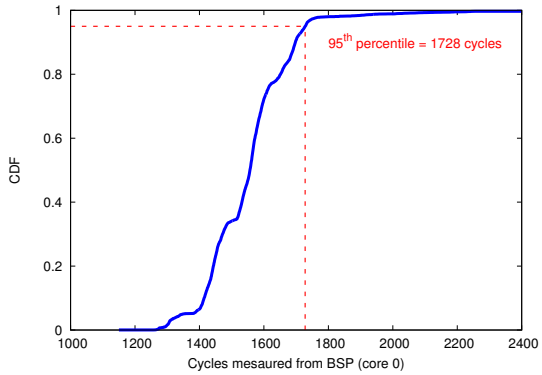
perscalar, but four hardware threads round-robin instruction-by-instruction for the execution core. As a consequence, the lower instruction count translates into a much lower average latency on the Phi.

The lower average context switch costs on the Phi translate directly into benefits for an HRT developer because it makes it feasible to more finely partition work. On both platforms, the lower variance makes more fine grain cooperation feasible. The default policies described in Section 2.1, combined with the performance characteristics shown here are intended to provide a predictable substrate for HRT development. The HRT developer can also readily override the default scheduling and binding model while still leveraging the fast thread creation/fork and context switch capabilities.

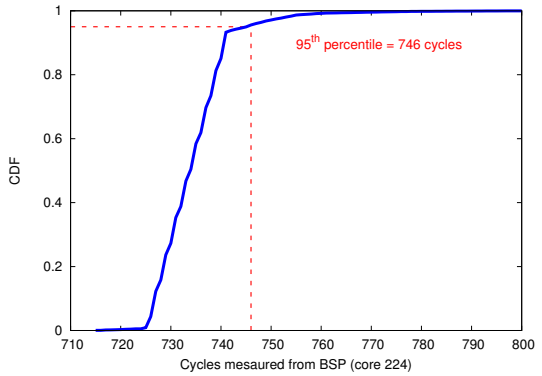
Events Figure 4 compares the event wakeup performance for the mechanisms discussed in Section 2.1 on the two platforms. We measure the latency from when an event is signaled to when the waiting thread executes. We compare the cost of condition variable wakeup in user mode in Linux with our two implementations of them (with and without IPI) in Aerokernel. We also show the performance of the Linux fast user space mutex ("futex") primitive, and of a oneway IPI, which is the hardware limit for an event wakeup.

For condition variables, the latency measured is from the call to `pthread_cond_signal` (or equivalent) and the subsequent wakeup from `pthread_cond_wait` (or equivalent). The IPI measurement is the time from when the IPI is initiated until when its interrupt handler on the destination hardware thread has written a memory location being monitored by the source hardware thread.

The average latency for Aerokernel's condition variables (with IPI) is five times lower than that of Linux user level on both platforms. It is also three to five times lower than the futex. Equally important, the variance in this latency is much



(a) x64



(b) phi

Figure 5: CDF of IPI one-way latencies, the hardware limit of asynchronous signaling that is available to HRTs.

lower on both platforms, by a factor of three to ten. From an HRT developer's perspective, these performance results mean that much "smaller" events or smaller units of work can feasibly be managed, and that these events and work can be more tightly synchronized in time.

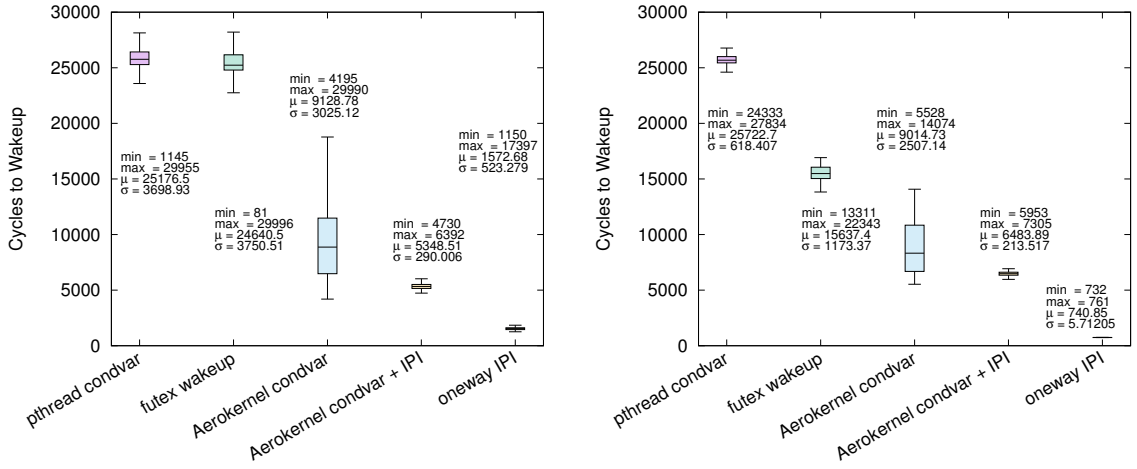
Because they operate in kernel mode, HRTs can make direct use of IPIs and thus operate at the hardware limit of asynchronous event notification, which is one to three thousand cycles on our hardware. Figure 5 illustrates the latency of IPIs, as described earlier, on our two platforms. The specific latency depends on which two cores are involved and the machine topology. This is reflected in the notches in the CDF curve. Note however that there is little variation overall—the 5^{th} and 95^{th} percentile are within hundreds of cycles.

4. Experiences in creating HRTs

We now describe our experience in using Aerokernel to transform parallel runtime systems into HRTs—to convert these user-level systems and their applications into kernels.

4.1 Legion

The Legion runtime system is designed to provide applications with a parallel programming model that maps well to



(a) x64 (b) phi

Figure 4: Event wakeup latency. Aerokernel conditional variable wakeup latency is on average five times faster than Linux (pthreads), and has 3–10 times less variation.

heterogeneous architectures [4, 66]. Whether the application runs on a single node or across nodes—even with GPUs—the Legion runtime can manage the underlying resources so that the application does not have to. Legion is of particular interest as an HRT because the primary focus of the Legion developers is on the design of the runtime system. This not only allows us to leverage their experience in designing runtimes, but also gives us access to a system designed with experimentation in mind. Further, the codebase has reached the point where the developers' ability to rapidly prototype new ideas is hindered by abstractions imposed by the OS.

Under the covers, Legion bears similarities to an operating system and concerns itself with issues that an OS must deal with, including task scheduling, isolation, multiplexing of hardware resources, and synchronization. The way that a complex runtime like Legion attempts to manage the machine to suit its own needs can often conflict with the services and abstractions provided by the OS.

As Legion is intended for heterogeneous hardware it is designed with a multi-layer architecture. It is split up into the *high-level* runtime and the *low-level* runtime. The high-level runtime is portable across machines, and the low-level runtime contains all of the machine-specific code. There is a separate low-level implementation called the *shared low-level runtime*. This is the low-level layer implemented for shared memory machines. All of our modifications to Legion when porting it to Aerokernel were made to this component. Outside of optimizations using hardware access, and understanding the needs of the runtime, the port was straight-forward.

Legion, in its default user-level implementation, uses pthreads as representations of logical processors, so the low-level runtime makes heavy use of the pthreads interface. We

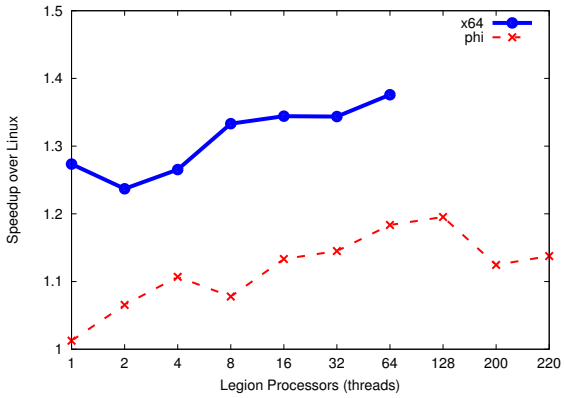


Figure 6: Aerokernel speedup of Legion HPCG (x64+phi).

created a variant that made use of the Aerokernel interface, particularly for threads and events. Our port is based on Legion as of October 2014 (commit e22962d), which can be found on legion.stanford.edu. The port to Aerokernel involved 1.5 person-months of effort, and approximately 200 lines of C/C++ code were added to Legion to support it. Probably the most complex part of the effort involved enhancing the building and linking logic so that the result was compatible with the Aerokernel model. The Legion distribution includes numerous test codes which we used to evaluate the correctness of our work.

To evaluate the performance benefits of applying the HRT model to Legion using Aerokernel, we used the HPCG (High Performance Conjugate Gradients) macrobenchmark. HPCG is an application benchmark effort from Sandia National Labs that is designed to help rank top-500 supercomputers for suitability to scalable applications of national interest [27, 38]. Los Alamos National Laboratory has ported

HPCG to Legion, and we used this port to further evaluate our Aerokernel variant of Legion. HPCG is a complex benchmark (∼5100 lines of C++) that exercises many Legion features. Recall that Legion itself comprises another ∼43,000 lines of C++.

Figure 6 shows the speedup of the HPCG/Legion in Aerokernel over HPCG/Legion on Linux as a function of the number of Legion processors being used. Each Legion processor is bound to a distinct hardware thread. On the Phi, Aerokernel is able to speed up HPCG by up to 20%. On x64, Aerokernel increases its performance by almost 40%. We configured HPCG for a medium problem size which, on a standard Linux setup, runs for roughly 2 seconds. We see similar results for other HPCG configurations.

There are many contributors to the increased performance of HPCG in Aerokernel, particularly fast condition variables. An interesting one is simply due to the simplified paging model. On x64 the Linux version exhibited almost 1.6 million TLB misses during execution. In comparison, the Aerokernel version exhibited about 100.

4.2 NESL

NESL [12] is a highly influential implementation of nested data parallelism developed at CMU in the '90s. Recently, it has influenced the design of parallelism in Manticore [34, 33], Data Parallel Haskell [20, 21], and arguably the nested call extensions to CUDA [56]. NESL is a functional programming language that allows the implementation of complex parallel algorithms in a compact and high-level way. NESL programs are compiled into abstract vector operations known as VCODE through a process known as flattening. An abstract VCODE interpreter then executes these programs on physical hardware. Flattening transformations and their ability to transform nested (recursive) data parallelism into "flat" vector operations while preserving the asymptotic complexity of programs is a key contribution of NESL [13] and recent work on using NESL-like nested data parallelism for GPUs [9] and multicore [8] has focused on extending flattening approaches to better match such hardware.

As a proof of concept, we ported NESL's existing VCODE interpreter to Aerokernel, allowing us to run any program compiled by the out-of-the-box NESL compiler. We also ported NESL's sequential implementation of the vector operation library CVL, which we have started parallelizing. Currently, vector operations with the exception of scans execute in parallel. The combination of the core VCODE interpreter and a CVL library form the VCODE interpreter for a system in the NESL model.

While this effort is a work in progress, it gives some insights into the challenges of porting this kind of parallel runtime to become an HRT. In summary, such a port is quite tractable. Our modifications to the NESL source code release[1] currently comprise about 100 lines of Makefile changes and 360 lines of C source code changes. About 220 lines of the C changes are in CVL macros that implement the point-wise vector operations we have parallelized using Aerokernel's thread fork/join facilities. The remainder (100 Makefile lines, 140 C lines) reflect the amount of glue logic that was needed to bring the VCODE interpreter and the serial CVL implementation into Aerokernel. The hardest part of this glue logic is assuring that the compilation and linking model match that of Aerokernel, which is reflected in the Makefile changes. The effort took about one person-month to bring to this point.

[1] Available from `www.cs.cmu.edu/~scandal/nesl/nesl3.1.html`

4.3 NDPC

We are creating a different implementation of a subset of the NESL language which we refer to as "Nested Data Parallelism in C/C++" (NDPC). This is implemented as a source-to-source translator whose input is the NESL subset and whose output is C++ code (with C bindings) that uses recursive fork/join parallelism instead of NESL's flattened vector parallelism. The C++ code is compiled directly to object code and executes without any interpreter or JIT. Because C/C++ is the target language, the resulting compiled NDPC program can easily be directly linked into and called from C/C++ codebases. NDPC's collection type is defined as an abstract C++ class, which makes it feasible for the generated code to execute over any C/C++ data structure provided it exposes or is wrapped with the suitable interface. We made this design decision to further facilitate "dropping into NDPC" from C/C++ when parallelism is needed. In the context of Figure 7, our intent is that the runtime processing of a call to an NDPC function will include crossing the boundary between the general purpose and specialized portions of the hybrid virtual machine.

The generated code uses a runtime that is written in C, C++, and assembly that provides preemptive threads and simple work stealing. Code generation is greatly simplified because the runtime supports a thread fork primitive. The runtime guarantees that a forked thread will terminate at the point it attempts to return from the current function. The NDPC compiler in turn guarantees the code it generates for the current function will only use the current caller and callee stack frames, that it will not place pointers to the stack *on* the stack, and that the parent will join with any forked children before it returns from the current function. The runtime's implementation of the thread fork primitive can thus avoid complex stack management. Furthermore, it can potentially provide fast thread creation, despite the fork semantics, because it can avoid most stack copying as only data on the caller and callee stack frames may be referenced by the child. In some cases, the compiler can determine the maximum stack size (e.g., for a leaf function), and supply this to the runtime, further speeding up thread creation.

The runtime supports being compiled to operate in user level, using pthreads or an internal fibers implementation, or to operate in kernel level using Aerokernel. The Aerokernel

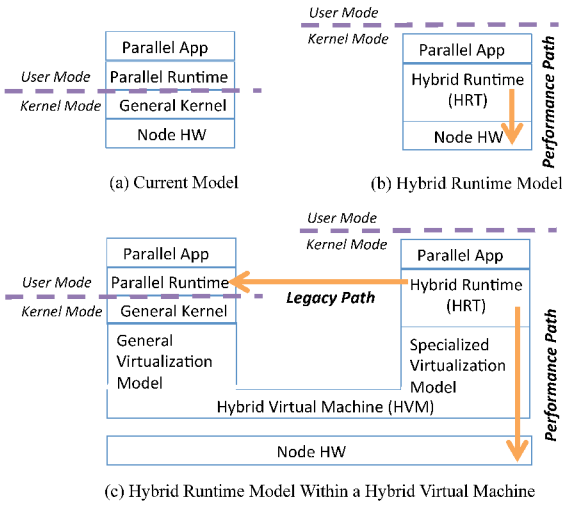


Figure 7: Overview of the HVM and the HRT.

support consists of only 150 lines of code. As with the NESL VCODE port (Section 4.2) the primary challenges in making NDPC operate at kernel-level within Aerokernel have to do with assuring that the compilation and linking models match those of Aerokernel. Currently, we are able to compile simple benchmarks such as nested data parallel quicksort into Aerokernel kernels and run them. NDPC is a work in progress, but the effort to bring it up in Aerokernel in its present state required about a person-week of effort.

5. Hybrid virtual machine

We currently envision four deployment models for an HRT:

- *Dedicated*: the machine is dedicated to the HRT, for example on a machine is an accelerator or a node of a supercomputer. This is the model used in the previous section.
- *Partitioned*: the machine is a supercomputer node that is physically partitioned [57], with a partition of cores dedicated to the HRT.
- *VM*: The machine's hypervisor creates a VM that runs the HRT.
- *HVM*: The machine's hypervisor creates a VM that is internally partitioned and runs both the HRT and a general purpose OS we call the "regular" OS (ROS).

For the *VM* and *HVM* deployment models, the hypervisor partitions/controls resources, and provides multiprogramming and access control. In the *HVM* model, the HRT can leverage both kernels, as we describe in detail here. Note that in an HVM, the ROS memory is vulnerable to modification by the HRT, but we think of the two as a unit; an unrelated process would be run in a separate VM (or on the host).

The purpose of the hybrid virtual machine (HVM) environment, illustrated in Figure 7, is to enable the execution of an HRT in a virtual environment simultaneously with the ROS. That is, a single VM is shared between a ROS and an HRT. The virtual environment exposed to the HRT may be different from and indeed much lower-level than the environment exposed to the ROS. It can also be rebooted independently of the ROS. At the same time, the HRT has access to the memory of the ROS and can interrupt it. In some ways, the HRT can be viewed as providing an "accelerator" for the ROS and its applications, and the HVM provides the functionality of bridging the two. Using the HVM, the performance critical elements of the parallel runtime can be moved into the HRT, while runtime functionality that requires the full stack remains in the ROS.

We have developed a prototype HVM environment in the context of the Palacios open source VMM. The concepts and prerequisites of our prototype are not specific to Palacios and could be readily implemented in other VMMs. Our prototype comprises about 3,500 lines of C and assembly.

5.1 Model

The user creates a virtual machine configuration, noting cores, memory, NUMA topology, devices, and their initial mappings to the underlying hardware. An additional configuration block specifies that this VM is to act as an HVM. The configuration contains three elements: (1) a partition of the cores of the VM into two groups: the HRT cores and the ROS cores, which will run their respective kernels; (2) a limit on how much of the VM's physical memory will be visible and accessible from a ROS core; and (3) a multiboot2-compliant kernel, such as an HRT built on top of Aerokernel. We extend the multiboot2 specification to support HRTs. The existence of a special header in the multiboot section of the ELF file indicates that the HRT boot model described here is to be followed instead of the standard multiboot2 model.

The remainder of the model can be explained by considering the view of a ROS core versus that of an HRT core. The VMM maintains the following invariants for a ROS core: (1) Only the portion of the VM's physical memory designated for ROS use is visible and accessible. (2) Inter-processor interrupts (IPIs) can be sent only to another ROS core. (3) Broadcast, group, lowest-priority, and similar IPI destinations consider only the ROS cores. An HRT core, on the other hand, has no such constraints. All of the VM's physical memory is visible and accessible. IPIs can be sent to any core, and broadcast, group, lowest-priority and similar IPI destinations can consider either all cores or only HRT cores. This is set by the HRT. Generally speaking, interrupts from interrupt controllers such as IOAPICs and physical interrupts can be delivered to both ROS and HRT cores. The default is that they are delivered only to ROS cores, but an HRT core can request them. Broadcast, group, lowest priority, and similar destinations are computed over the ROS cores and any requesting HRT cores.

5.2 Protection

The invariants described above are implemented through two techniques. The memory invariants leverage the fact that

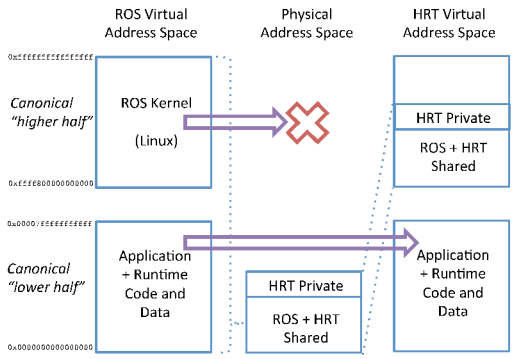


Figure 8: Merged address space.

each core in the VMM we use, similar to other VMMs, maintains its own paging state, for example shadow or nested page tables. This state is incrementally updated due to exits on the core, most importantly due to page faults or nested page faults. For example, a write to a previously unmapped address causes an exit during which we validate the access and add the relevant page table entries in the shadow or nested age tables if it is an appropriate access. A similar validation occurs for memory-mapped I/O devices. We have extended this model so that validation simply takes into account the type of core (ROS or HRT) on which the exit occurred. This allows us to catch and reject attempts by a ROS core to access illegal guest physical memory addresses.

The interrupt invariants are implemented by extensions to the VMM's APIC device. At the most basic level, IPI delivery from any APIC, IOAPIC, or MSI takes into account the type of core from which the IPI originates (if any) and the type of core that it targets. IPIs from a ROS core to an HRT core are simply dropped. External interrupts targeting an HRT core are delivered only if previously requested. Additionally, the computations involved with broadcast and group destinations for IPIs and external interrupts are modified so that only cores that are prospective targets are included. Similarly, the determination of the appropriate core or cores for a lowest priority destination includes only those cores to which the interrupt could be delivered under the previous restrictions. These mechanisms allow us to reject any attempt to send an HRT core an unwanted interrupt.

5.3 Merged address space

The HRT can access the entire guest physical address space, and thus can operate directly on any data within the ROS. However, to simplify the creation of the legacy path shown in Figure 7, we provide the option to merge an address space within the ROS with the address space of the HRT, as is shown in Figure 8. When a merged address space is in effect, the HRT can use the same user-mode virtual addresses that are used in the ROS. For example, the parallel runtime in the ROS might load files and construct a pointer-based data structure in memory. It could then invoke a function within its counterpart in the HRT to operate on that data.

Item	Cycles	Time
Address Space Merger	∼33 K	15 μs
Asynchronous Call	∼25 K	11 μs
Synchronous Call (different socket)	∼1060	482 ns
Synchronous Call (same socket)	∼790	359 ns

Figure 9: Round-trip latencies of ROS↔HRT interactions (x64).

To achieve this we leverage the canonical 64-bit address space model of x64 processors, and its wide use within existing kernels, such as Linux. In this model, the virtual address space is split into a "lower half" and a "higher half" with a gap in between, the size of which is implementation dependent. In a typical process model, e.g., Linux, the lower half is used for user addresses and the higher half is used for the kernel.

For an HRT that supports it, the HVM arranges so that the physical address space is identity-mapped into the higher half of the HRT address space. That is, within the HRT, the physical address space mapping (including the portion of the physical address space only the HRT can access) occupies the same portion of the virtual address space that is occupied by the ROS kernel, the higher half. Without a merger, the lower half is unmapped and the HRT runs purely out of the higher half. When a merger is requested, we map the lower half of the ROS's current process's address space into the lower half of the HRT address space. For a Aerokernel-based HRT, this is done by copying the first 256 entries of the PML4 from the PML4 pointed to by the ROS's CR3 to the HRT's PML4 and then broadcasting a TLB shootdown to all HRT cores.

Because the parallel runtime in the ROS and the HRT are co-developed, the responsibility of assuring that page table mappings exist for lower half addresses used by the HRT in a merged address space is the parallel runtime's. For example, the parallel runtime can pin memory before merging the address spaces, or introduce a protocol to send page faults back to the ROS. The former is not an unreasonable expectation in a high performance environment as we would never expect to be swapping.

5.4 Communication

The HVM model makes it possible for essentially any communication mechanism between the ROS and HRT to be built, and most of these require no specific support in the HVM. As a consequence, we minimally defined the *basic* communication between the ROS, HRT, and the VMM using shared physical memory, hypercalls, and interrupts.

The user-level code in the ROS can use hypercalls to sequentially request HRT reboots, address space mergers, and asynchronous sequential or parallel function calls. The VMM handles reboots internally, and forwards the other two requests to the HRT as interrupts. Because additional information may need to be conveyed, a data page is shared

between the VMM and the HRT. For a function call request, the page essentially contains a pointer to the function and its arguments at the start and the return code at completion. For an address space merger, the page contains the CR3 of the calling process. The HRT indicates to the VMM when it is finished with the current request via a hypercall.

After an address space merger, the user-level code in the ROS can also use a single hypercall to initiate synchronous operation with the HRT. This hypercall ultimately indicates to the HRT a virtual address which will be used for future synchronization between the HRT and ROS. A simple memory-based protocol can then be used between the two to communicate, for example for the ROS to invoke functions in the HRT, without VMM intervention.

Figure 9 shows the measured latency of each of these operations, using Aerokernel as the HRT.

5.5 Boot and reboot

The ROS cores follow the traditional PC bootstrap model with the exception that the ACPI and MP tables built in memory show only the hardware deemed visible to the ROS by the HVM configuration.

Boot on an HRT core differs from both the ROS boot sequence and from the multiboot2 specification [58], which we leverage. Multiboot2 for x86 allows for bootstrap of a kernel into 32-bit protected mode on the first core (the BSP) of a machine. Our extension allows for bootstrap of a kernel in full 64-bit mode. There are two elements to HRT boot—memory setup and core bootstrap. These elements combine to allow us to *simultaneously* start all HRT cores immediately at the entry point of the HRT. At the time of this startup, each core is running in long mode (64-bit mode) with paging and interrupt control enabled. The HRT thus does not have much bootstrap to do itself. A special multiboot tag within the kernel indicates compatibility with this mode of operation and includes requests for how the VMM should set up the kernel environment.

In memory setup, which is done only once in the lifetime of the HRT portion of the VM, we select an HRT-only portion of the guest physical address space and lay out the basic machine data structures needed: an interrupt descriptor table (IDT) along with dummy interrupt and exception handlers, a global descriptor table (GDT), a task state segment (TSS), and a page table hierarchy that identity-maps physical addresses (including the higher-half offset as shown in Figure 8, if desired) using the largest feasible page table entries. We also select an initial stack location for each HRT core. A simple ELF loader then copies the HRT ELF into memory at its desired target location. Finally, we build a multiboot2 information structure in memory. This structure is augmented with headers that indicate our variant of multiboot2 is in use, and provide fundamental information about the VM, such as the number of cores, the APIC IDs, interrupt vectoring, and the memory map, including the areas containing the memory

Item	Cycles (and exits)	Time
HRT core boot of Aerokernel to `main()`	∼135 K (7 exits)	61 μs
Linux `fork()`	∼320 K	145 μs
Linux `exec()`	∼1 M	476 μs
Linux `fork()` + `exec()`	∼1.5 M	714 μs
HRT core boot of Aerokernel to idle thread	∼37 M (∼2300 exits)	17 ms

Figure 10: HRT reboot latencies in context (x64).

setup. Because bootstrap occurs on virtual hardware this information can be much simpler than that supplied via ACPI.

In core bootstrap, which may be done repeatedly over the lifetime of the HRT portion of the HVM, the registers of the core are set. The registers that must be set include the control registers (IDTR, GDTR, LDTR, TR, CR0, CR3, CR4, EFER), the six segment registers including their descriptor components, and the general purpose registers RSP, RBP, RDI, and RAX. The point is that core bootstrap simply involves setting about 20 register values. The instruction pointer (RIP) is set to the entry point of the HRT, while RSP and RBP are set to the initial stack for the core, and RDI points to the multiboot2 header information and RAX contains the multiboot2 cookie.

Unlike a ROS boot, all HRT cores are booted together simultaneously. The HRT is expected to synchronize these internally. In practice this is easy as a core can quickly find its rank by consulting its APIC ID and looking at the APIC ID list given in the extended multiboot2 information.

Fast HRT Reboot Because core bootstrap involves changing a small set of registers and then reentering the guest, the set of HRT cores can be rebooted very quickly. An HRT reboot is also independent of the execution of the ROS, and an HRT can be therefore be rebooted many times over the lifetime of the HVM. We allow an HRT reboot to be initiated from the HRT itself, from a userspace utility running on the host operating system, and via a hypercall from the ROS, as described above.

Figure 10 illustrates the costs of rebooting an HRT core, and compares it with the cost of typical process operations on a Linux 2.6.32 kernel running on the same hardware. An HRT core can be booted and execute to the first instruction of Aerokernel's `main()` in ∼50% of the time it takes to do a Linux process `fork()`, ∼13% of the time to do a Linux process `exec()` and ∼8% of the time to do a combined `fork()` and `exec()`. The latter is the closest analog in Linux to what the HRT reboot accomplishes. Note also that timings on Linux were done "hot"—executables were already memory resident.

A complete reboot of Aerokernel on the HRT core to the point where the idle thread is executing takes 17 ms. This time is also blindingly fast compared to the familiar norm of booting a physical or virtual machine. We anticipate that

this time will further improve for two reasons. First, we can in principle skip much of the general purpose startup code in Aerokernel, which is currently executed, given that we know exactly what the virtual hardware looks like. Second, by starting the core from a memory and register snapshot, specifically at the point of execution we desire to start from, we should be able to even further short-circuit startup code.

It is important to note that even at 17 ms, a complete Aerokernel reboot is 60 to 300 times faster than a typical 1-5 minute node or server boot time. It should be thought of in those terms, similar to the MicroReboot concept [18] for cheap recovery from software failures. We can use HRT reboots to address many issues and, in the limit, treat them as being on par with process creation in a traditional OS.

6. Related work

The design of Aerokernel was influenced by early research on microkernels [51, 11, 10] and even more by Engler and others' work on exokernels [29, 30]. Using exokernel terminology, Aerokernel can be thought of as a kind of library OS for a parallel runtime, but we shed the notion of privilege levels for the sake of functionality and performance. Other important OS projects in the vain of thin kernel layers include KeyKOS [14], ADEOS [69], and the Stanford Cache Kernel [25]. More recently there has been a resurgence of ideas from exokernel in the context of virtualization. Dune uses hardware virtualization support to allow applications to have access to a certain protected hardware features [7]. Arrakis leverages virtualized I/O devices in a similar vain in order to allow hardware access [59]. OSv [45], Unikernels [53], and the Drawbridge and Bascule libOSes [60, 6] are other examples. OSv, for example, does not eliminate the user/kernel distinction. Aerokernel is unique in that it is designed to support the hybrid runtime model, giving parallel runtimes unfettered access to the full feature set of the machine. Aerokernel is conceptually similar to Libra [1], but Libra does not provide a notion of a large shared address space between Libra/J9 and the Linux support VM. Furthermore, Aerokernel does not require HVM capability in order to run. That is, it does not rely on Palacios as an exokernel layer. For example, we can quickly boot a hybrid runtime instance on raw Xeon Phi hardware. Aerokernel's fast bootstrap capability exists independent of the HVM environment. Linux containers [54] provide fast-launching virtual instances as well, but they still maintain a user/kernel distinction and do not allow the use of a specialized kernel.

Aerokernel bears some similarity to other single address space OSes (SASOSes), including Opal [24], Singularity [42], Scout [55], and Nemesis [61]. Aerokernel targets single-node performance, particularly for many-core machines. We therefore drew inspiration from some notable projects with similar goals, including Barrelfish [5], Tesselation OS [52], Corey [15], K42 [47], and, of course, work on scaling Linux [16]. PTask [62] provides kernel-level abstractions for GPUs, including a data flow abstraction that can be used by the kernel itself. None of this work explicitly shapes an OS around the needs of parallel runtime systems. As far as we are aware, this is a unique property of Aerokernel.

The HPC community has long felt that OSes "get in the way". Ferreira and Hoefler both explored the performance impact of OS noise on large-scale parallel applications [31, 32, 40]. Lightweight kernels such as Kitten [48] and mOS [68] attempt to mitigate the problem, but granting runtimes fully privileged access is not one of the solutions explored. There has been a decades-long interest in bridging the gap between complex hardware and the programmer through languages and runtime systems which is now seeing a resurgence in the exascale space. Languages and language implementations coming from the HPC community, such as OpenARC [50], Chapel [22], UPC [19], CoArray Fortran [28], and X10 [23] could be users of the hybrid runtime concept. Swift [46]'s model of many tiny tasks is of particular resonance. Another common thread in bridging the gap between complex hardware and the programmer is to enhance program and runtime execution by manipulating the system from user level. COSMIC [17] targets the Intel Phi, while Juggle [41] targets NUMA machines. The HRT model allows direct control of decisions that such systems can only encourage.

Other approaches to realizing the split-machine model shown in Figure 7 exist. Dune, described above, provides one alternative. Guarded modules [36] could be used to give portions of a general-purpose virtualization model selective privileged access to hardware, including I/O devices. Pisces [57] would enable an approach that could eschew virtualization altogether by partitioning the hardware and booting multiple kernels simultaneously without virtualization.

7. Conclusion

We introduced the hybrid runtime (HRT) model, in which a parallel runtime system and its application are transformed into a specialized OS kernel that can take direct advantage of all hardware features. Two core tools, the Nautilus Aerokernel and HVM, enable the model. Aerokernel provides a suite of functionality specialized to HRT development that can perform up to two orders of magnitude faster than the general purpose functionality in the Linux kernel while also providing much less variation in performance. Aerokernel functionality leads to 20-40% performance gains in an application benchmark for the Legion runtime system on x64 and Xeon Phi. HVM is VMM functionality that allows us to simultaneously run two kernels, an HRT and a traditional kernel, within the same VM, allowing a runtime to benefit from the performance and capabilities provided by the HRT model while not losing the performance non-critical functionality of the traditional kernel.

References

[1] AMMONS, G., APPAVOO, J., BUTRICO, M., DA SILVA, D., GROVE, D., KAWACHIYA, K., KRIEGER, O., ROSENBURG, B., HENSBERGEN, E. V., AND WISNIEWSKI, R. W. Libra: A library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3^{rd} International Conference on Virtual Execution Environments (VEE 2007)* (June 2007), pp. 44–54.

[2] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13^{th} ACM Symposium on Operating Systems Principles (SOSP 1991)* (Oct. 1991), pp. 95–109.

[3] BAE, C., LANGE, J., AND DINDA, P. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8^{th} International Conference on Autonomic Computing (ICAC 2011)* (June 2011).

[4] 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).

[5] BAUMANN, A., BARHAM, P., DAGAND, P. E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22^{nd} ACM Symposium on Operating Systems Principles (SOSP 2009)* (Oct. 2009), pp. 29–44.

[6] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8^{th} ACM European Conference on Computer Systems (EuroSys 2013)* (Apr. 2013), pp. 239–252.

[7] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10^{th} USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)* (Oct. 2012), pp. 335–348.

[8] BERGSTROM, L., FLUET, M., RAINEY, M., REPPY, J., ROSEN, S., AND SHAW, A. Data-only flattening for nested data parallelism. In *Proceedings of the 18^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013)* (Feb. 2013), pp. 81–92.

[9] BERGSTROM, L., AND REPPY, J. Nested data-parallelism on the GPU. In *Proceedings of the 17^{th} ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)* (Sept. 2012), pp. 247–258.

[10] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15^{th} ACM Symposium on Operating Systems Principles (SOSP 1995)* (Dec. 1995), pp. 267–283.

[11] BLACK, D. L., GOLUB, D. B., JULIN, D. P., RASHID, R. F., DRAVES, R. P., DEAN, R. W., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., AND BOHMAN, D. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (Apr. 1992), pp. 11–30.

[12] 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.

[13] BLELLOCH, G. E., AND GREINER, J. A provable time and space efficient implementation of NESL. In *Proceedings of the 1^{st} ACM SIGPLAN International Conference on Functional Programming (ICFP 1996)* (May 1996), pp. 213–225.

[14] BOMBERGER, A. C., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures* (Apr. 1992), pp. 95–112.

[15] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8^{th} USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)* (Dec. 2008), pp. 43–57.

[16] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An analysis of Linux scalability to many cores. In *Proceedings of the 9^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010)* (Oct. 2010).

[17] CADAMB, S., COVIELLO, G., LI, C.-H., PHULL, R., RAO, K., SANKARADASS, M., AND CHAKRADHAR, S. COSMIC: Middleware for high performance and reliable multiprocessing on xeon phi coprocessors. In *Proceedings of the 22^{nd} ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2013)* (June 2013), pp. 215–226.

[18] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot: A technique for cheap recovery. In *Proceedings of the 6^{th} USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)* (Dec. 2004), pp. 31–44.

[19] CARLSON, W., DRAPER, J., CULLER, D., YELICK, K., BROOKS, E., AND WARREN, K. Introduction to upc and language specification. Tech. Rep. CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[20] CHAKRAVARTY, M., KELLER, G., LESHCHINSKIY, R., AND PFANNENSTIEL, W. Nepal—nested data-parallelism in haskell. In *Proceedings of the 7^{th} International Euro-Par Conference (EUROPAR 2001)* (Aug. 2001).

[21] 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).

[22] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the chapel langauge. *International Journal of High Performance Computing Applications 21*, 3 (Aug. 2007), 291–312.

[23] CHARLES, P., DONAWA, C., EBICIOGLU, K., GROTHOFF, C., KIELSTRA, A., VON PRAUN, C., SARASWAT, V., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the* 20^{th} *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)* (Oct. 2005), pp. 519–538.

[24] CHASE, J. S., LEVY, H. M., LEVY, H. M., FEELEY, M. J., FEELEY, M. J., LAZOWSKA, E. D., AND LAZOWSKA, E. D. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems 12*, 4 (Nov. 1994), 271–307.

[25] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proceedings of the* 1^{st} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)* (Nov. 1994).

[26] DAS, R., UYSAL, M., SALTZ, J., AND HWANG, Y.-S. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing 22*, 3 (September 1994), 462–478.

[27] DONGARRA, J., AND HEROUX, M. A. Toward a new metric for ranking high performance computing systems. Tech. Rep. SAND2013-4744, Sandia National Laboratories, June 2013.

[28] DOTSENKO, Y., COARFA, C., AND MELLOR-CRUMMEY, J. A multi-platform co-array fortran compiler. In *Proceedings of the* 13^{th} *International Conference on Parallel Architectures and Compilation Techniques (PACT 2004)* (Sept. 2004), pp. 29–40.

[29] ENGLER, D. R., AND KAASHOEK, M. F. Exterminate all operating system abstractions. In *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems (HotOS 1995)* (May 1995), pp. 78–83.

[30] 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* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP 1995)* (Dec. 1995), pp. 251–266.

[31] 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).

[32] 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).

[33] FLUET, M., RAINEY, M., REPPY, J., AND SHAW, A. Implicitly threaded parallelism in manticore. In *Proceedings of the* 13^{th} *ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)* (Sept. 2008), pp. 119–130.

[34] 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 (DAMP 2007)* (Jan. 2007), pp. 37–44.

[35] GOERBESSIOTIS, A. V., AND VALIANT, L. G. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing 22*, 2 (1994), 251–267.

[36] HALE, K. C., AND DINDA, P. A. Guarded modules: Adaptively extending the VMM's privilege into the guest. In *Proceedings of the* 11^{th} *International Conference on Autonomic Computing (ICAC 2014)* (June 2014), pp. 85–96.

[37] HALE, K. C., AND DINDA, P. A. A case for transforming parallel runtimes into operating system kernels. In *Proceedings of the* 24^{th} *International Symposium on High-performance Parallel and Distributed Computing (HPDC 2015)* (June 2015), pp. 27–32.

[38] HEROUX, M. A., DONGARRA, J., AND LUSZCZEK, P. HPCG technical specification. Tech. Rep. SAND2013-8752, Sandia National Laboratories, October 2013.

[39] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 2.0. Tech. rep., Center for Research on Parallel Computation, Rice University, January 1996.

[40] 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).

[41] HOFMEYR, S., COLMENARES, J. A., IANCU, C., AND KUBIATOWICZ, J. Juggle: Proactive load balancing on multicore computers. In *Proceedings of the* 20^{th} *ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2011)* (June 2011), pp. 3–14.

[42] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review 41*, 2 (Apr. 2007), 37–49.

[43] KAISER, H., BRODOWICZ, M., AND STERLING, T. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the* 38^{th} *International Conference on Parallel Processing Workshops (ICPPW 2009)* (Sept. 2009), pp. 394–401.

[44] 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.

[45] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014)* (June 2014).

[46] KRIEDER, S., WOZNIAK, J., ARMSTRONG, T., WILDE, M., KATZ, D., GRIMMER, B., FOSTER, I., AND RAICU, I. Design and evaluation of the GeMTC framework for gpu-enabled many-task computing. In *Proceedings of the* 23^{rd} *ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2014)* (June 2014), pp. 153–164.

[47] KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATER-

LAND, A., AND UHLIG, V. K42: Building a complete operating system. In *Proceedings of the 1^{st} ACM European Conference on Computer Systems (EuroSys 2006)* (Apr. 2006), pp. 133–145.

[48] 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 24^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)* (Apr. 2010).

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

[50] LEE, S., AND VETTER, J. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23^{rd} ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2014)* (June 2014), pp. 115–120.

[51] LIEDTKE, J. On micro-kernel construction. In *Proceedings of the 15^{th} ACM Symposium on Operating Systems Principles (SOSP 1995)* (Dec. 1995), pp. 237–250.

[52] LIU, R., KLUES, K., BIRD, S., HOFMEYR, S., ASANOVIĆ, K., AND KUBIATOWICZ, J. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1^{st} USENIX Conference on Hot Topics in Parallelism (HotPar 2009)* (Mar. 2009).

[53] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)* (Mar. 2013), pp. 461–472.

[54] MENAGE, P. B. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium* (June 2007), pp. 45–58.

[55] MONTZ, A. B., MOSBERGER, D., O'MALLEY, S. W., PETERSON, L. L., AND PROEBSTING, T. A. Scout: A communications-oriented operating system. In *Proceedings of the 5^{th} Workshop on Hot Topics in Operating Systems (HotOS 1995)* (May 1995), pp. 58–61.

[56] NVIDIA CORPORATION. Dynamic parallelism in CUDA, Dec. 2012.

[57] OAYANG, J., KOCOLOSKI, B., LANGE, J., AND PEDRETTI, K. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24^{th} International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC 2015)* (June 2015), pp. 149–160.

[58] OKUJI, Y. K., FORD, B., BOLEYN, E. S., AND ISHIGURO, K. The multiboot specification—version 1.6. Tech. rep., Free Software Foundation, Inc., 2010.

[59] PETER, S., AND ANDERSON, T. Arrakis: A case for the end of the empire. In *Proceedings of the 14^{th} Workshop on Hot Topics in Operating Systems (HotOS 2013)* (May 2013).

[60] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. In *Proceedings of the 16^{th} International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)* (Mar. 2011), pp. 291–304.

[61] ROSCOE, T. Linkage in the Nemesis single address space operating system. *ACM SIGOPS Operating Systems Review 28*, 4 (Oct. 1994), 48–55.

[62] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 2011)* (2011).

[63] 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 1^{st} ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC 2012)* (Sept. 2012).

[64] 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).

[65] TEW, K., SWAINE, J., FLATT, M., FINDLER, R., AND DINDA, P. Places: Adding message passing parallelism to racket. In *Proceedings of the 7^{th} Dynamic Languages Symposium (DLS 2011)* (Oct. 2011), pp. 85–96.

[66] 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.

[67] WHEELER, K. B., MURPHY, R. C., AND THAIN, D. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the 22^{nd} International Symposium on Parallel and Distributed Processing (IPDPS 2008)* (Apr. 2008).

[68] WISNIEWSKI, R. W., INGLETT, T., KEPPEL, P., MURTY, R., AND RIESEN, R. 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)* (June 2014).

[69] YAGHMOUR, K. Adaptive domain environment for operating systems. `http://www.opersys.com/ftp/pub/Adeos/adeos.pdf`.