NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
Number: NU-EECS-15-01

April 1, 2014

Details of the Case for Transforming Parallel Runtime Systems Into Operating System Kernels

Kyle C. Hale and Peter A. Dinda

Abstract

The needs of parallel runtime systems and the increasingly sophisticated languages and compilers they support do not line up with the services provided by general-purpose OSes. Furthermore, the semantics available to the runtime are lost at the system-call boundary in such OSes. Finally, because a runtime executes at user-level in such an environment, it cannot leverage hardware features that require kernel-mode privileges---a large portion of the functionality of the machine is lost to it. These limitations warp the design, implementation, functionality, and performance of parallel runtimes. We make the case for eliminating these compromises by transforming parallel runtimes into OS kernels. We also demonstrate that it is feasible to do so. Our evidence comes from Nautilus, a prototype kernel framework that we built to support such transformations. After describing Nautilus, we report on our experiences using it to transform three very different runtimes into kernels.

This project is made possible by support from the United States National Science Foundation (NSF) via grant CNS- 0709168, the Department of Energy (DOE) via grant DE- SC0005343, and 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 DOE Office of Science. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Keywords: hybrid runtimes, HRTs, hybrid virtual machine, HVM, Nautilus

Details of the Case for Transforming Parallel Run-times Into Operating System Kernels

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

ABSTRACT

The needs of parallel run-time systems and the increasingly sophisticated languages and compilers they support do not line up with the services provided by general-purpose OSes. Furthermore, the semantics available to the run-time are lost at the system-call boundary in such OSes. Finally, because a run-time executes at user-level in such an environment, it cannot leverage hardware features that require kernel-mode privileges—a large portion of the functionality of the machine is lost to it. These limitations warp the design, implementation, functionality, and performance of parallel run-times. We make the case for eliminating these compromises by transforming parallel run-times into OS kernels. We also demonstrate that it is feasible to do so. Our evidence comes from Nautilus, a prototype kernel framework that we built to support such transformations. After describing Nautilus, we report on our experiences using it to transform three very different run-times into kernels.

1. INTRODUCTION

Modern parallel run-times are systems that operate in user mode and run above the system call interface of a general-purpose kernel. While this facilitates portability and simplifies the creation of some functionality, it also has consequences that warp the design and implementation of the run-time and affect its performance, efficiency, and scalability. First, the run-time is deprived of the use of hardware features that are available only in kernel mode. This is a large portion of the machine. For example, approximately 1/3 to 1/2 of the Intel processor architecture manual content deals with such features. The second consequence is that the run-time must use the abstractions provided by the kernel, even if the abstractions are a bad fit. For example, the run-time might need subset barriers, and be forced to build them out of mutexes. Finally, the kernel has minimal access to the information available to the parallel run-time or to the language implementation it supports. For example, the run-time might not require coherence, but get it anyway.

The complexity of modern hardware is rapidly growing. In high-end computing, it is widely anticipated that exascale machines will have at least 1000-way parallelism at the node level. Even today's high-end homogeneous nodes, such as the one we use for evaluation in this paper, have 64 or more cores or hardware threads arranged on top of a complex intertwined cache hierarchy that terminates in 8 or more memory zones with non-uniform access. Today's heterogeneous nodes include accelerators, such as the Intel Phi, that introduce additional coherence domains and memory systems. Server platforms for cloud and datacenter computing, and even desktop and mobile platforms are seeing this simultaneous explosion of hardware complexity and the need for parallelism to take advantage of the hardware. How to make such complex hardware programmable, in parallel, by mere humans is an acknowledged open challenge.

Some modern run-times, such as the Legion run-time [2, 49] we consider in this paper, already address this challenge by creating abstractions that programmers or the compilers of high-level languages can target, abstractions that mirror the machine in portable ways. Very high-level parallel languages can let us further decouple the expression of parallelism from its implementation. Parallel run-times such as Legion, and the run-times for specific parallel languages share many properties with operating system (OS) kernels, but suffer by not *being* kernels. With current developments, particularly in virtualization and hardware partitioning, we are in a position to remove this limitation. In this paper, we make the case for transforming parallel run-time systems into kernels, and report on our initial results with a framework to facilitate just that.

We argue that for the specific case of a parallel run-time, the user/kernel abstraction itself, which dates back to Multics, is not a good one. It's important to understand the kernel/user abstraction and its implications. This abstraction is an incredibly useful technique to enforce isolation and protection for processes, both from attackers and from each other. This not only enables increased security, but also reduces the impact of bugs and errors on the part of the programmer. Instead, programmers place a higher level of trust in the kernel, which, by virtue of its smaller codebase and careful design, ensures that the machine remains uncompromised. However, because the kernel must be all things to all processes, the kernel has grown dramatically bigger over time, as has its responsibilities within the system. This has forced kernel developers to provide a broad range of services to an even broader range of applications. At the same time,

the basic model and core services have necessarily ossified in order to maintain compatibility with the widest range of hardware and software. In a general-purpose kernel, the needs of parallelism and a parallel run-time have not been first-order concerns.

Run-time implementors often complain about the limitations imposed by a general-purpose kernel. While there are many examples of significant performance enhancements within general-purpose kernels, and others are certainly possible to support parallel run-times better, a parallel run-time as a user level component is fundamentally *constrained* by the kernel/user abstraction. In contrast, as a kernel, a parallel run-time would have full access to all hardware features of the machine, and the ability to create any abstractions that it needs using those features. We show in this paper that, in fact, breaking free from the user/kernel abstraction can produce measurable benefits for parallel run-times.

At first glance, transforming a parallel run-time into a kernel seems to be a particularly daunting task because language run-times often have many dependencies on libraries and system calls. It is important to be clear that we are focused on the performance or energy-critical core of the run-time where the bulk of execution time is spent, not on the whole functional base of the run-time. The core of the run-time has considerably fewer dependencies and thus is much more feasible to transform into a kernel. As we describe in Section 2, virtualization and hardware partitioning in various forms have the potential to allow us to partition the run-time so the non-core elements run at user-level on top of the full software stack they expect, while the core of the run-time runs as a kernel. We refer to such a kernel as a hybrid run-time (HRT) as it is a hybrid between a kernel and a run-time. Our focus in this paper is on the HRT.

We make the following contributions:

- We describe the limitations of building parallel run-time systems on top of general-purpose operating systems and how these limitations are avoided if the run-time *is* a kernel. That is, we motivate HRTs.
- We describe the design, implementation, and performance of Nautilus, a prototype kernel framework to facilitate the porting of existing parallel run-times to run as kernels, as well as the implementation of new parallel run-times directly as kernels. That is, we create the tools needed to easily make HRTs.
- We describe our experiences in using Nautilus to transform three run-times into kernels, specifically Legion, NESL, and a new language implementation named NDPC that is being co-developed with Nautilus. That is, we make HRTs, demonstrating their feasibility.

2. ARGUMENT

A language's run-time is a system (typically) charged with two major responsibilities. The first is allowing a program written in the language to interact with its environment (at runtime). This includes access to underlying software layers (e.g., the OS) and the machine itself. The run-time abstracts the properties of both and impedance-matches them

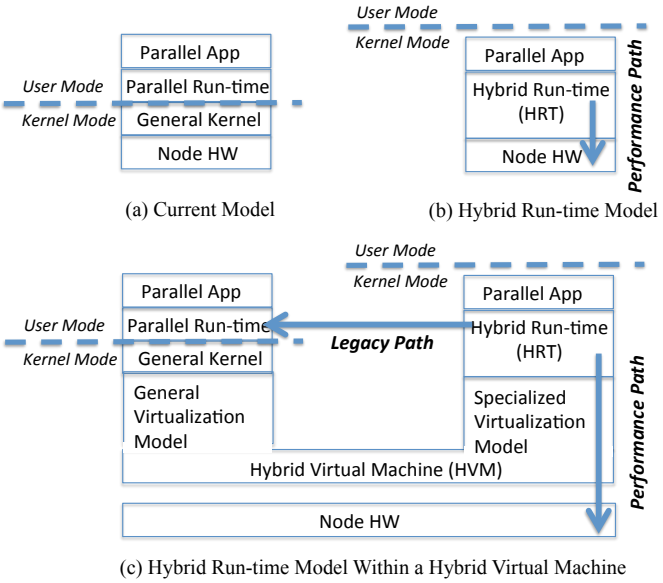

Figure 1: Overview of Hybrid Run-time (HRT) approach: (a) current model used by parallel run-times, (b) proposed HRT model, and (c) proposed HRT model combined with a hybrid virtual machine (HVM).

with the language's model. The challenges of doing so, particularly for the hardware, depend considerably on just how high-level the language is—the larger the gap between the language model and the hardware and OS models, the greater the challenge. At the same time, however, a higher-level language has more freedom in implementing the impedance-matching.

The second major responsibility of the run-time is carrying out tasks that are hidden from the programmer but necessary to program operation. Common examples include garbage collection in managed languages, JIT compilation or interpretation for compilers that target an abstract machine, exception management, profiling, instrumentation, task and memory mapping and scheduling, and even management of multiple execution contexts or virtual processors. While some run-times may offer more or less in the way of features, they all provide the programmer with a much simpler view of the machine than if he were to program it directly.

As a run-time gains more responsibilities and features, the lines between the run-time and the OS often become blurred. For example, Legion manages execution contexts (an abstraction of cores or hardware threads), regions (an abstraction of NUMA and other complex memory models), task to execution context mapping, task scheduling with preemption, and events. In the worst case this means that the run-time and the OS are actually trying to provide the *same* functionality. In fact, what we have found is that in some cases this duplication of functionality is brought about by inadequacies of or grievances with the OS and the services it provides. A common refrain of run-time developers is that they want the kernel to simply give them a subset of the machine's resources and then leave them alone. They attempt to approximate this as best they can within the confines of user space and the available system calls.

That this problem would arise is not entirely too surprising. After all, the operating system is, *prima facie*, designed to provide adequate performance for a broad range of general-purpose applications. However, when applications demand more control of the machine, the OS can often get in the way, whether due to rigid interfaces or to mismatched priorities in the design of those interfaces. Not only may the kernels abstractions be at odds with the run-time, it may also completely prevent the run-time from using hardware features that might otherwise significantly improve performance or functionality. If it provides access to these features, it does so through a system call, which—even if it has an appropriate interface for the run-time—nonetheless exacts a toll for use, as the system call mechanism itself has a cost. Similarly, even outside system calls, while the kernel might build an abstraction on top of a fast hardware mechanism, an additional toll is taken. For example, signals are simply more expensive than interrupts, even if they are used to abstract an interrupt.

A run-time that *is* a kernel will have none of these issues. It would have full access to all hardware features of the machine, and the ability to create any abstractions that it needs using those features. We want to support the construction of such run-times, which we call Hybrid Run-times (HRTs), as they are hybrids of parallel run-times and kernels. To do so, we will provide basic kernel functionality on a take-it-or-leave-it basis to make the process easier. We also want such run-time kernels to have available the full functionality of the general-purpose OS for components not central to run-time operation.

Figure 1 illustrates three different models for supporting a parallel run-time system. The current model (a) layers the parallel run-time over a general-purpose kernel. The parallel run-time runs in user mode without access to privileged hardware features and uses a kernel API designed for general-purpose computations. In the Hybrid Run-time model (b) that we describe in this paper the parallel run-time is integrated with a specialized kernel framework such as Nautilus. The resulting HRT runs exclusively in kernel mode with full access to all hardware features and with kernel abstractions designed specifically for it. Notice that both the run-time and the parallel application itself are now below the kernel/user line. Figure 1(b) is how we run Legion, NESL, and NDPC programs in this paper. We refer to this as the *performance path*.

A natural concern with the structure of Figure 1(b) is how to support general-purpose OS features. For example, how do we open a file? We do not want to reinvent the wheel within an HRT or a kernel framework such as Nautilus in order to support kernel functionality that is not performance critical. Figure 1(c) is our response, the Hybrid Virtual Machine (HVM). In an HVM, the virtual machine monitor (VMM) or other software will partition the physical resources provided to a guest, such as cores and memory into two parts. One part will support a general purpose virtualization model suitable for executing full OS stacks and their applications, while the second part will support a virtualization model specialized to the HRT and allowing it direct hardware access. The specialized virtualization model will enable the performance path of the HRT, while the general virtualization model and communication between the two parts of the HVM will enable a *legacy path* for the run-time and application that will let it leverage the capabilities of the general-purpose kernel for non-performance critical work.

An effort to build this HVM capability into the Palacios VMM [38] is currently underway in our group as part of the Hobbes exascale software project [15]. However, it is important to note that other paths exist. For example, Guarded Modules [31] could be used to give portions of a general-purpose virtualization model selective privileged access to hardware, including I/O devices. As another example, Dune [5] uses hardware virtualization features to provide privileged CPU access to Linux processes. The HVM could be built on top of Dune. The Pisces system [45] would enable an approach that could eschew virtualization altogether by partitioning the hardware and booting multiple kernels simultaneously without virtualization. Our focus in this paper is not on the HVM capability, but rather on the HRT.

3. NAUTILUS

Nautilus[1] is a small prototype kernel framework that we built to support the HRT model, and is thus the first of its kind. We designed Nautilus to meet the needs of parallel run-times that may use it as a starting point for taking full advantage of the machine. We chose to minimize imposition of abstractions to support general-purpose applications in lieu of flexibility and small codebase size. As we will show in Sections 4–6, this allowed us to port three very different run-times to Nautilus and the HRT model in a very reasonable amount of time. Nautilus is a Multiboot2-compliant kernel and we have tested it on several Intel and AMD machines, as well as QEMU and our own Palacios VMM.

As Nautilus is a prototype for HRT research, we targeted the most popular architecture for high-performance and parallel computing, x86_64. However, given the very tractable size of the codebase, introducing platform portability would not be too challenging. A port to the Intel Phi is underway.

We stress that the design of Nautilus is, first and foremost, driven by the needs of the parallel run-times that use it. Nevertheless, it is complete enough to leverage the full capabilities of a modern 64-bit x86 machine to support three run-times, one of which (Legion) is quite complex and is used in practice today.

3.1 Design and Performance

Currently, Nautilus is designed to boot the machine, discover its capabilities, devices, and topology, and immediately hand control over to the run-time. Figure 2 shows the core functionality provided by Nautilus. Current features include multi-core support, Multiboot2-compliant modules, synchronization primitives, threads, IRQs, timers, paging, NUMA awareness, IPIs, and a console. We will describe these capabilities in turn. We spent a good deal of time measuring the capabilities that affect the performance of the HRTs we built, and we will report on their performance in the following paragraphs.

[1]Named after the submarine-like, mysterious vessel from Jules Verne's Twenty Thousand Leagues Under the Sea.

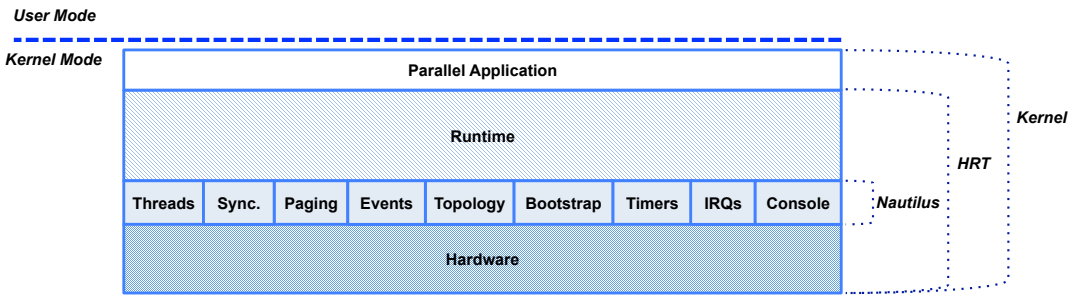

Figure 2: Structure of Nautilus.

OS	Avg	Min	Max
Nautilus	16795	2907	44264
Linux	38456	34447	238866

Figure 3: Time to create a single thread measured in cycles.

Experimental setup We took all measurements on our lab machine named *leviathan*. We chose this machine for our experiment because it has a large number of cores and an interesting organization, 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. 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 with stock Linux kernel version 2.6.32.

Threads In designing a threading model for Nautilus, we considered the experiences of many others, including work on high-performance user-level threading techniques like scheduler activations [1] and Qthreads [50]. Ultimately, we designed our threads to be very lightweight in order to provide an efficient starting point for HRTs. The threading model is not *imposed* on the run-time. It is simply offered as a primitive. We found that our threads performed quite well compared to traditional user-space pthread usage. It is important to note that, unlike pthreads, the threads we use in Nautilus are *kernel* threads. They are more than that however, because there is *only* a kernel, which includes the HRT and the Nautilus kernel framework. The nature of the threads in Nautilus 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) 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.

Figure 3 shows the average, minimum, and maximum time in cycles to create a single thread in both Nautilus and Linux (using pthreads). These numbers were taken over 1000 runs and we used the `rdtscp` instruction to enforce proper serialization of instructions when timing. Notice how the minimum time to create a thread for Nautilus is about 11x faster than with pthreads.

Figure 4 shows the time to create a number of threads in sequence, and we take results over 10 runs. This figure illustrates that the performance advantage of Nautilus's very light-weight threads, in fact, increases as the thread count scales. This creates an advantage for parallel run-times that need to leverage node-level parallelism to create units of work very quickly. Graphs (d), (e), and (f) highlight the performance difference as a function of thread count by showing the speedup of Nautilus's threading facilities over pthreads in Linux. We believe these results show that Nautilus provides a reasonable starting point for HRTs attempting to exploit the full potential of the machine.

Another distinctive aspect to Nautilus 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 run-time to manage the parent and child stacks correctly. This capability is leveraged in our ports of NESL and NDPC.

Synchronization We now give a brief description of the simple spinlock primitive used in Nautilus. We chose to highlight spinlocks because in designing Nautilus, we focused heavily on the Legion run-time's model of execution, in which threads represent logical processors. The ideal case for Legion occurs when threads are pinned to CPUs and experience no contention (i.e. no preemption) for physical resources. In this case, more complex synchronization primitives like mutexes are unnecessary.

Nautilus's implementation of a spinlock uses a GCC intrinsic that compiles down into an atomic `xchg` instruction, which will enforce locking on the memory bus. We use the `pause` instruction in between spins to allow maximum performance for hyperthreads on the same physical core. This instruction will insert a small delay into the spin loop, freeing up pipeline resources for potential contenders.

Figure 5 shows the total time to acquire and subsequently release a spinlock 500 million times on both Nautilus and Linux. Here, "Linux" means we use `pthread_spin_lock`. Note that the time comes within 9% of the heavily optimized pthreads version. The bottom rows show the time, measured in cycles, to acquire and release a single lock one time. The

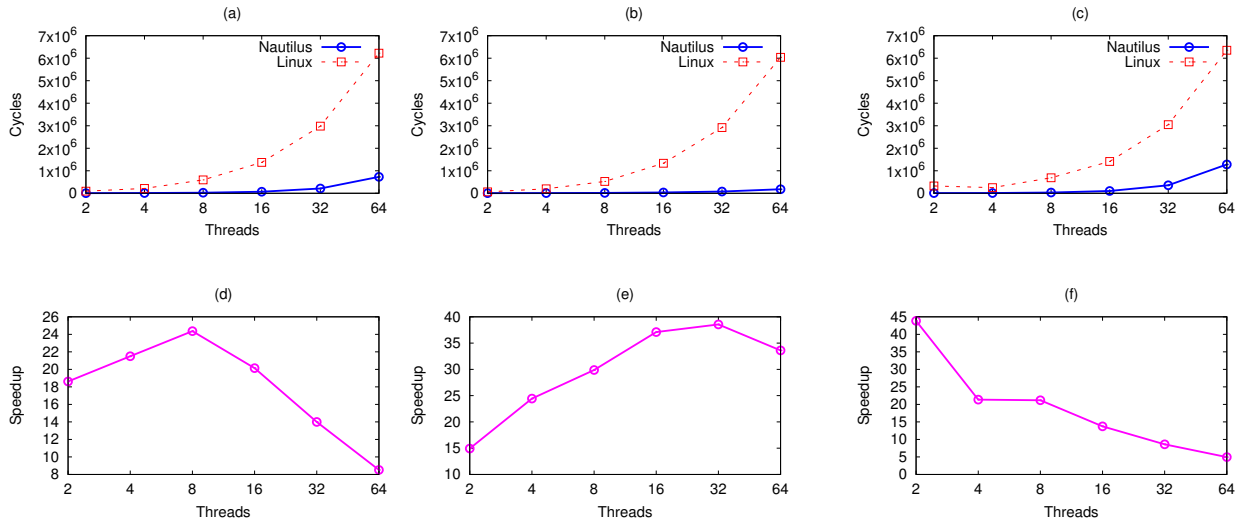

Figure 4: Average (a), minimum (b), and maximum (c) time to create a number of threads in sequence. Average (d), minimum (e), and maximum (f) speedup of Nautilus over Linux for multiple thread creations.

OS	**Execution time (s)**
Nautilus	13.72
Linux	12.53
OS	**Avg. acquire/release time (cycles)**
Nautilus	59
Linux	36

Figure 5: Total time to acquire and release a spinlock 500 million times on Nautilus and Linux, and average time in cycles for an acquire/release pair.

parallel applications we tested on Legion spend most of their time in computation-heavy loops, and are thus not heavily influenced by the cost of synchronization.

Paging Nautilus has a very simple, yet very high-performance paging model aimed at high-performance parallel applications. When the machine boots up, each core identity-maps the entire physical address space using 2 MB pages to create a single *unified* address space. The static identity map eliminates expensive page faults and TLB shootdowns, and reduces TLB misses. These events would not only reduce performance, but also introduce unpredictable OS noise. 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 run-times, but also limits the *kinds* of run-times that can take advantage of the machine. For example, run-times that need tasks to execute in synchrony (e.g., in order to support a bulk-synchronous parallel application or a run-time 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 very fast communication between threads, and eliminates much

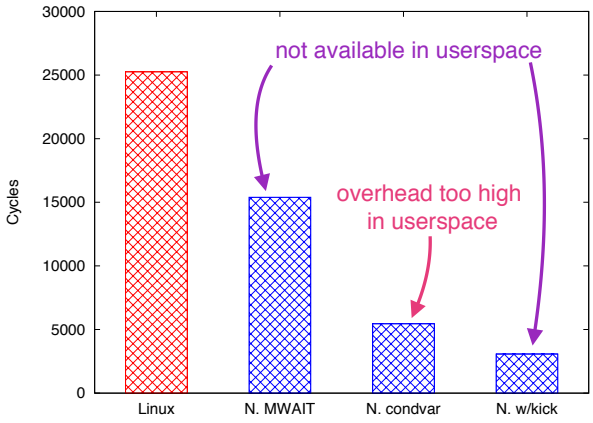

Figure 6: Average event wakeup latency.

of the overhead of context switches when Nautilus boots with preemption enabled. The only preemption is between kernel threads, so no page table switch ever occurs. This is especially useful when Nautilus 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 Nautilus exhibits the minimum possible shadow page faults, and shadow paging can be more efficient that nested paging, except when shadow page faults are common.

Events Events are a common abstraction that run-time systems often use to distribute work to execution units, or workers. The Legion run-time makes heavy use of them, so we wanted to make sure that Nautilus provided an efficient implementation of them. In Legion, the events are used to notify logical processors (Legion threads) when there are *tasks* ready to execute. To help show the potential of Legion + Nautilus as an HRT, we measured the performance of these "wakeup" events.

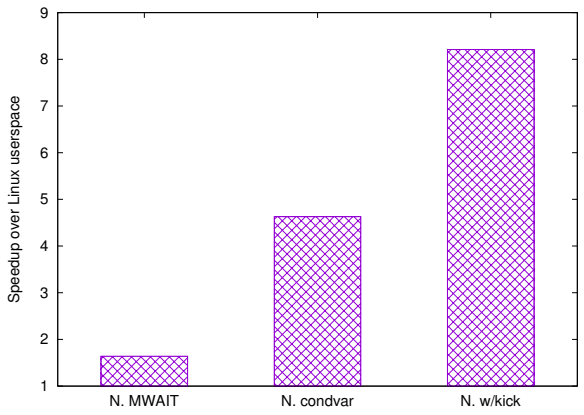

Figure 7: Speedup over Linux for event wakeups in Nautilus.

Figure 6 shows the average latency between an event notification and the subsequent wakeup. Here, we had a single thread on one core go to sleep and wait for an event notification from a thread running on the adjacent physical core. The latency is measured in cycles and the average is taken over 100 runs. The first box on the left shows the latency of a common mechanism used in Linux for event notification, the pthread implementation of condition variables. In this case, the measurement is the time between calling `pthread_cond_signal` and the subsequent wakeup from `pthread_cond_wait`. A wakeup takes about 25000 cycles. The following three boxes show various Nautilus implementations of event notification. "N.MWAIT" shows the latency when using the newer `MONITOR/MWAIT` extensions provided by modern processors. These instructions allow one thread to go to sleep on a range of memory, waiting for a write to that memory by another thread. Note that the `MONITOR/MWAIT` instructions are *not* available in user-space. While the latency improves considerably over pthread's condition variables in Linux user-space, we suspect it is limited by the hardware latency incurred when waking up from a sleep state. The `MONITOR/MWAIT` extensions provide optional hints to enter lower sleep states and allow faster wakeups, but our machine does not support this feature.

The final two boxes ("N. condvar" and "N. w/kick") show the Nautilus implementation of condition variables. They are very lightweight, and a signal will essentially just enqueue the waiting thread on a processor's run queue. This is another capability that is not typically available to a user-space thread. At some point, the kernel's scheduler must intervene to take the thread off of a wait queue and schedule it. In Nautilus, the run-time can do this directly. Notice the significant latency improvement in the Nautilus implementations over pthreads. Figure 7 shows the improvement in terms of speedup.

When a thread is signalled in Nautilus, if the next timer tick does not occur immediately, the thread may have some delay before execution begins. To mitigate this potential issue, we introduced an optimization to Nautilus condition variables that "kicks" the appropriate CPU by sending it an inter-processor interrupt (IPI) after it has woken up the waiting thread. An IPI sends a message over an interrupt delivery network connecting the interrupt controllers of the processor cores. The core specified in the destination address will wake up (if needed) and immediately execute the corresponding interrupt handler. This gives the run-time the ability to force execution of a tasks of its choosing on a remote destination core. Note that the IPI is unavailable when running in user-space. The performance is shown in the last box of Figure 6. Notice how the wakeup latency is reduced by about 36% from the default condition variable implementation. This tells us that IPIs have a great deal of potential as a hardware feature that HRTs can leverage.

Target	**Min. latency (cycles)**
Rem. Socket 1	4135
Rem. Socket 2	4161
Rem. Socket 3	3926
Loc. Socket, Rem. Node	3902
Loc. Socket, Rem. Core	3707
Loc. Socket, Loc. Core	4691

Figure 8: IPI round-trip latencies on *leviathan* with varying distances.

Event notifications can also be implemented using messages, and IPIs are a very useful messaging transport—the Linux kernel uses them in many places. To explore the potential of run-time event notifications using IPIs, we measured their round-trip latency on our *leviathan* machine. Figure 8 shows the minimum latency of 100 round-trip IPIs between various locations within the machine. We measure a round-trip latency by reading the TSC, sending an IPI to a remote core, and waiting for its acknowledgement. When it arrives, we read the TSC again to produce a latency in cycles. Here, "Rem. Node" means an IPI to the same socket but to a core on a remote NUMA node. "Loc. Socket, Rem. Core" indicates a core on the same socket *and* NUMA node. The last row shows the latency to a hardware thread on the same physical core. Local cores within the same chip are closest and thus have the smallest round-trip latency at about 3700 cycles, or 1.8μsec. From the inter-socket measurements, we suspect that socket 1 is the furthest away from and socket 3 is the closest to the sending core. The IPI latency to a hardware thread on the same physical core is surprisingly high, and we suspect this may be due to contention over resources they share. These measurements also line up with those in Figure 6, as the cost of a wakeup is roughly the same as an IPI round-trip latency to a core on the same socket.

Toplogy Large, modern machines typically organize memory into separate domains according to physical distance from a physical CPU socket or core. This introduces a variable latency when accessing memory in the different domains. This variable latency is typically referred to as Non-Uniform Memory Access (NUMA). Platform firmware typically enumerates these NUMA domains and exposes their sizes and toplogy to the operating system in a way that supports both modern and legacy OSes.

In order to provide maximum performance to parallel run-times, Nautilus includes support for NUMA machines. While this may seem like a small issue, in practice we saw NUMA effects that would double the execution time of a long-running parallel application on the Legion run-time. NUMA-awareness is therefore very important. While user-space processes do typically have access to NUMA information and policies, run-times executing in the Nautilus framework have *full* con-

Language	SLOC
C	22697
C++	133
x86 Assembly	428
Scripting	706

Figure 9: Source lines of code for the Nautilus kernel.

trol over the placement of threads and memory and can thus enjoy guarantees about what can affect run-time performance.

Bootstrap Nautilus bootstraps the machine much like any other OS, but is spared the need to bring up a complex user-space environment. Once Nautilus initializes the physical resources of the machine, such as memory, processors, and devices, control over these resources falls into the hands of the run-time. Nautilus enforces no policies or mechanisms that may limit the performance or functionality of parallel run-times. Because HRT bootstrap may well take the place of process creation within an HVM, the timing of Nautilus bootstrap is important. On our hardware, the time from when the boot loader invokes Nautilus to the time the first instruction of the run-time executes is on the order of a few milliseconds.

Timers By default, Nautilus enables a scheduler tick mechanism so that run-times may, if they require it, implement a preemptive scheduling model. The default periodic timer interrupt in Nautilus comes from the Advanced Programmable Interrupt Controller (APIC) present on all modern x86 machines. We chose the APIC timer because every processor core has its own APIC timer, and therefore does not need to receive scheduling events from other cores. This frees up IPIs for other uses within the run-time.

For timing of events, Nautilus provides primitives to read several platform timers, including the legacy i8254 PIT and the more precise high-precision event timer (HPET). For example, Nautilus exposes a `clock_gettime()` function that will read the HPET's monotonic counter registers. While this made the process of porting run-times easier, the run-time is by no means limited to using this method.

IRQs External interrupts in Nautilus work just like any other operating system, with the exception that by default only the timer interrupt is enabled at bootup. The run-time has complete control over interrupts, including their mapping, assignment, and priority ordering.

Console Nautilus exposes a set of text-mode console utilities (such as `printk()`) that allow a run-time to immediately display useful output on the machine. This output can be routed to the video card and/or a serial port.

3.2 Complexity

We now make a case for the potential for Nautilus as a vehicle for HRTs, now setting aside the attractive performance of its primitives.

The process of building Nautilus as a minimal kernel layer with support for a complex, modern, many-core x86 machine took six person-months of effort on the part of seasoned OS/VMM kernel developers. Figure 9 shows that Nautilus is fairly compact, with a total of roughly 24000 lines of code.

Language	SLOC
C++	133
C	636

Figure 10: Lines of code added to Nautilus to support Legion, NDPC, and NESL.

Building a kernel, however, was not our main goal. Our main focus was supporting the porting and construction of run-times for the HRT model. The Legion run-time, discussed at length in the next section, was the most challenging and complex of the three run-times to bring up in Nautilus. Legion is about double the size of Nautilus in terms of codebase size, consisting of about 43000 lines of C++. Porting Legion and the other run-times took a total of about four person-months of effort. Most of this work went into understanding Legion and its needs. The lines of code actually added to Nautilus to support all three run-times is shown in Figure 10. We only needed to add about 800 lines of code. This is tiny considering the size of the Legion run-time.

This suggests that exploring the HRT model for existing or new parallel run-times, especially with a small kernel like Nautilus designed with this in mind, is a perfectly manageable task for an experienced systems researcher or developer. We hope that these results will encourage others to similarly explore the benefits of HRTs.

4. EXAMPLE: LEGION

The Legion run-time system is designed to provide applications with a parallel programming model that maps well to heterogeneous architectures [2, 49]. Whether the application runs on a single node or across nodes—even with GPUs—the Legion run-time can manage the underlying resources so that the application does not have to. There are several reasons why we chose to port Legion to the HRT model. The first is that the primary focus of the Legion developers is on the design of the run-time system. This not only allows us to leverage their experience in designing run-times, 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 layer. Another reason we chose Legion is that it is quickly gaining adoption among the HPC community, including within the DOE's exascale effort. The third reason is that we have corresponded directly with the Legion developers and discussed with them issues with the OS layer that they found when developing their run-time.

Under the covers, Legion bears many similarities to an operating system and concerns itself with many issues that an OS must deal with, including task scheduling, isolation, multiplexing of hardware resources, and synchronization. As we discussed in Section 2, the way that a complex run-time 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 designed for heterogeneous hardware, including multi-node clusters and machines with GPUs, it is designed with a multi-layer architecture. It is split up into the

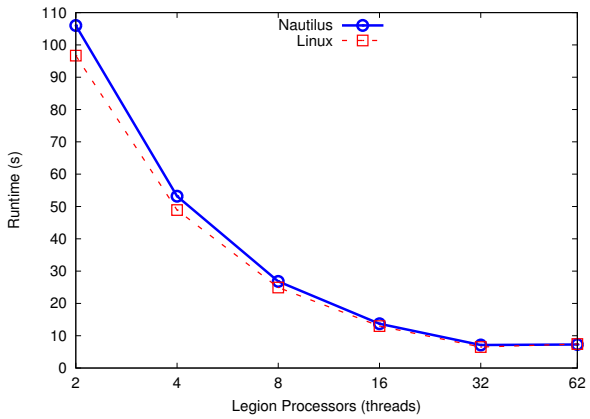

Figure 11: Run time of Legion circuit simulator versus core count. The baseline Nautilus version has higher performance at 62 cores than the Linux version.

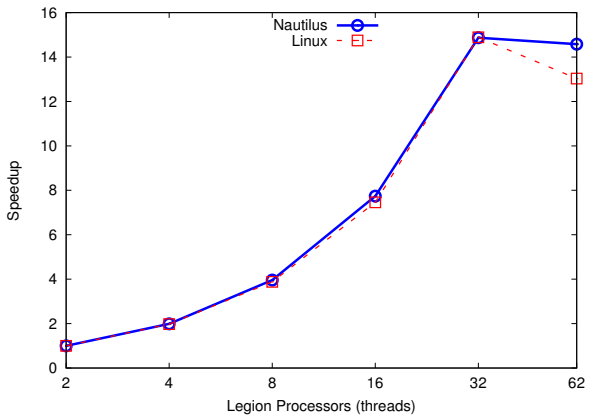

Figure 12: Speedup of Legion (normalized to 2 Legion processors) circuit simulator running on Linux and Nautilus as a function of Legion processor (thread) count.

high-level run-time and the *low-level* run-time. The high-level run-time is portable across machines, and the low-level run-time contains all of the machine specific code. There is a separate low-level implementation called the *shared low-level run-time*. This is the low-level layer implemented for shared memory machines. As we are interested in single-node performance, we naturally focused our efforts on the shared low-level Legion run-time. All of our modifications to Legion when porting it to Nautilus were made to the shared low-level component. Outside of optimizations using hardware access, and understanding the needs of the run-time, the port was fairly straight-forward.

Legion, in its default user-level implementation, uses pthreads as representations of logical processors, so the low-level run-time makes fairly heavy use of the pthreads interface. In order to transform Legion into a kernel-level HRT, we simply had to provide a similar interface in Nautilus. The amount of code added to Nautilus was less than 800 lines, and is described in Figure 10. After porting Legion into Nautilus, we then began to explore how Legion could benefit from unrestricted access to the machine.

We now evaluate our transformation of the user-level Legion run-time into a kernel using Nautilus, highlighting the realized and potential benefits of having Legion operate as an HRT. Our port is based on Legion as of October 14, 2014, specifically commit e22962dbc05e52897a3c699085df9ad19590453a, which can be found via the Legion project web site.[2]

The Legion distribution includes numerous test codes, as well as an example parallel application that is a circuit simulator. We used the test codes to check the correctness of our work and the circuit simulator as our initial performance benchmark. Legion creates an abstract machine that consists of a set of cooperating threads that execute work when it is ready. These are essentially logical processors. The number of such threads can vary, representing an abstract machine of a different size.

We ran the circuit simulator with a medium problem size (100000 steps) and varied the number of cores Legion used

[2]http://legion.stanford.edu

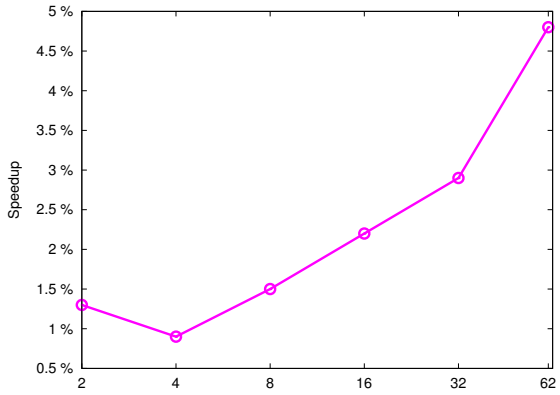

Figure 13: Speedup of Legion circuit simulator comparing the baseline Nautilus version and a Nautilus version that executes Legion tasks with interrupts off.

to execute Legion tasks. Figure 11 shows the results. The x-axis shows the number of threads/logical processors. The thread count only goes up to 62 because the Linux version would hang at higher core counts, we believe due to a live-lock situation in Legion's interaction with Linux. Notice how closely, even with no hardware optimizations, Nautilus tracks the performance of Linux. The difference between the two actually increases when scaling the number of threads. They are essentially at parity, even though Nautilus and the Legion port to it are still in their early stages. Nautilus is slightly faster at 62 cores.

The speedup of the circuit simulator running in Legion as a function of the number of cores is shown in Figure 12. Speedups are normalized to Legion running with two threads. The circuit simulator is largely CPU-bound and spends 99% of its time in a loop computing a stencil approximation for a PDE. We suspect that the benefits of running in an HRT would be magnified in a more memory-bound code or one that stresses the system primitives (see Section 3).

To experiment with hardware functionality in the HRT model, we wanted to take advantage of a capability that normally isn't available in Linux at user-level. We decided to use the capability to disable interrupts. In the Legion HRT, there

Language	SLOC
Compiler	
Lisp	11005
Run-time	
C	8853
lex	230
yacc	461

Figure 14: Source lines of code for NESL. The run-time consists of the VCODE interpreter and the CVL implementation we use.

are no other threads running besides the threads that Legion creates, and so there is really no need for timer interrupts (or device interrupts). Observing that interrupts can cause interference effects at the level of the instruction cache and potentially in task execution latency, we inserted a call to disable interrupts when Legion invokes a task (in this case the task to execute a function in the circuit simulator). Figure 13 shows the results, where the speedup is over the baseline case where Legion is running in Nautilus but without any change in the default interrupt policy. While this extremely simple change involved only adding two short lines of code, we can see a measurable benefit that scales with the thread count, up to 5% at 62 cores.

We believe that this is a testament to the promise of the HRT model. While the Legion port to Nautilus is still in its early stages, there is a large opportunity for exploring other potential hardware optimizations to improve run-time performance.

5. EXAMPLE: NESL

NESL [10] is a highly influential implementation of nested data parallelism developed at CMU in the '90s. Very recently, it has influenced the design of parallelism in Manticore [30, 29], Data Parallel Haskell [19, 20], and arguably the nested call extensions to CUDA [44]. NESL is a functional programming language, using an ML-style syntax that allows the implementation of complex parallel algorithms in a very compact and high level way, often 100s or 1000s of times more compactly than using a low-level language such as C+OpenMP. NESL programs are compiled into abstract vector operations known as VCODE through a process known as flattening. An abstract machine called a 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 [11] and very recent work on using NESL-like nested data parallelism for GPUs [7] and multicore [6] has focused on extending flattening approaches to better match such hardware.

As a proof of concept, we have ported NESL's existing VCODE interpreter to Nautilus, allowing us to run any program compiled by the out-of-the-box NESL compiler in kernel mode on x86_64 hardware. We also ported NESL's sequential implementation of the vector operation library CVL, which we have started parallelizing. Currently, point-wise vector operations 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.

Language	SLOC
Compiler	
Perl	2000
lex	82
yacc	236
Run-time	
C	2900
C++	93
x86 assembly	477

Figure 15: Source lines of code for NDPC. The run-time counts include code both for use on Nautilus and for user-level use.

While this effort is a work in progress, it gives some insights into the challenges of porting this form of language implementation to the kernel level. In summary, such a port is quite tractable. A detailed breakdown of the source code in NESL as we use it is given in Figure 14. Compared to the NESL source code release[3], our modifications 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. 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 Nautilus. The hardest part of this glue logic is assuring that the compilation and linking model match that of Nautilus, which is reflected in the Makefile changes. The effort took about one quarter to complete.

6. EXAMPLE: NDPC

We have created 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 1(c), we plan that the run-time 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.

Figure 15 breaks down the NDPC implementation in terms of the languages used and the size of the source code in each. The NDPC compiler is written in Perl and its code breaks down evenly between (a) parsing and name/type unification, and (b) code generation. The code generator can produce sequential C++ or multithreaded C++. The generated code uses a run-time that is written in C, C++, and assembly that provides preemptive threads and simple work stealing.

[3] http://www.cs.cmu.edu/~scandal/nesl/nesl3.1.html

Code generation is greatly simplified because the run-time supports a `thread_fork()` primitive. The run-time guarantees that a forked thread will terminate at the point it attempts to return from the current function. The NDPC compiler 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 run-time's implementation of `thread fork()` can thus avoid complex stack management. Furthermore, it can potentially provide very 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 run-time, further speeding up thread creation.

We also note that the compiler knows exactly what parts of the stack can be written, and it knows that the lifetime of a child thread nests within the lifetime of its parent. This knowledge could be potentially leveraged at the kernel level by maintaining only a single copy of read-only data on the parent stack and the child stacks.

Two user-level implementations of threading are included in the run-time, one of which is a veneer on top of pthreads, while the other is an implementation of user-level fibers that operates similarly to a kernel threading model. The kernel-level implementation, for Nautilus, consists of only 150 lines of code, as Nautilus supports this threading model internally. It is important to point out that the thread fork capability was quite natural to add to Nautilus, but somewhat painful to graft over pthreads. Even for user-level fibers, the thread fork capability requires somewhat ugly trampoline code which is naturally avoided in the kernel.

As with the NESL VCODE port (Section 5) the primary challenges in making NDPC operate at kernel-level within Nautilus have to do with assuring that the compilation and linking models match those of Nautilus. An additional challenge has been dealing with C++ in the kernel, although the C++ feature set used by code generated by NDPC is considerably simpler than that needed by Legion. Currently, we are able to compile simple benchmarks such as nested data parallel quicksort into Nautilus kernels and run them. NDPC is a work in progress, but the effort to bring it up in Nautilus in its present state required about a week.

7. RELATED WORK

The design of Nautilus was heavily influenced by early research on microkernels [40, 9, 8] and even more by Engler and others' work on exokernels [26, 27]. Like Nautilus, exokernel promotes an extremely thin kernel layer that only serves to provide isolation and basic primitives. Higher-level abstractions are delegated to user-mode *library OSes*. Nautilus can be thought of as a kind of library OS for a parallel run-time, but we shed the notion of privilege levels for the sake of functionality and performance. Other important OS projects in the vain of very thin kernel layers include KeyKOS [12], ADEOS [52], and the Stanford Cache Kernel [24]. In many cases this idea has become intertwined with modern virtualization [16]. More recently there has been a resurgence of ideas from exokernel. Dune uses hardware virtualization support to allow applications to have access to a certain protected hardware features [5]. Arrakis leverages virtualized I/O devices in a similar vain in order to allow hardware access [46]. OSv [35], Unikernels [42], and the Drawbdridge and Bascule libOSes [47, 4] also use virtualization to shed more light on the potential of the exokernel idea. Nautilus is unique, however, in that it is designed to support the hybrid run-time model, giving the run-time unfettered access to the full feature set of the machine.

There is also more evidence in the HPC community that the OS can "get in the way". Ferreira and Hoefler both explored the performance impact of OS noise on large-scale parallel applications [28, 32]. Light-weight kernels such as Kitten [38] are one solution to this problem. Current efforts are underway to address this issue and work towards a solution at the OS level [51], but granting access to run-times is not one of the solutions explored.

The HPC community has had a decades-long interest in bridging the gap between complex hardware and the programmer through languages and run-time systems. This is now becoming a serious challenge at the node level for exascale, a challenge that was the focus of the SC 2013 panel on "Exascale Run-time Systems". The SC 2014 panel on "Changing Operating Systems is Harder than Changing Programming Languages" followed the earlier discussion. Languages and language implementations coming from the HPC community, such as OpenARC [39], Chapel [21], UPC [18], CoArray Fortran [25], and X10 [22] have the potential for bridging this gap and could be users of the hybrid run-time concept. As a specific example, consider the Swift language and run-time [36], which seeks to support a programming model in which there are many tiny tasks. Starting a task must have extremely low overhead, kernel-level operation could bolster. Another common thread in bridging the gap between complex hardware and the programmer is to enhance program and run-time execution by manipulating the system from user level. COSMIC [17], for example, targets the Intel Phi, while Juggle [33] targets NUMA machines. The HRT model would allow direct control of decisions that systems like these can only encourage indirectly.

Nautilus also bears some similarity to other single address space OSes (SASOSes), including Opal [23], Singularity [34], Scout [43], and Nemesis [48]. The design choice of a single address space for Nautilus was made mostly for simplicity, but we plan to explore other alternatives in future work.

Nautilus targets single-node performance, particularly for many-core machines. We therefore drew inspiration from some notable projects with similar goals, including Barrelfish [3], Tesselation OS [41], Corey [13], K42 [37], and, of course, work on scaling Linux [14]. None of this work, however, explicitly shapes an OS around the needs of parallel run-time systems. As far as we are aware, this is a unique property of Nautilus.

8. CONCLUSIONS AND FUTURE WORK

We have made the case for transforming parallel run-times into operating system kernels, forming hybrid run-times (HRTs).

The motivations for HRTs include the increasing complexity of hardware, the convergence of parallel run-time concerns and abstractions in managing such hardware, and the limitations of executing the run-time at user-level, both in terms of limited hardware access and limited control over kernel abstractions. We presented and evaluated Nautilus, a prototype kernel framework to facilitate the construction of HRTs. Using Nautilus, we were able to successfully transform three very different run-times into HRTs. For the Legion run-time, we were able to exceed Linux performance with simple techniques that can only be done in the kernel. Building Nautilus was a six person-month effort, while porting the run-times was a four person-month effort. It is somewhat remarkable that even with a fairly nascent kernel framework, *just* by dropping the run-time down to kernel level and taking advantage of a kernel-only feature in two lines of code, we can exceed performance on Linux, an OS that has undergone far more substantial development and tuning effort.

In this paper, we have motivated HRTs, demonstrated tools to make them a reality, and made several HRTs. We are currently investigating other run-times, other architectures (particularly Intel Phi), and are examining the use of other kernel-only hardware features. A subtle point is that the run-times we have ported thus far still bear much resemblance to their original userspace counterparts—that is, they are ports instead of new designs developed without the constraints of userspace. We also note that the performance results given Section 4 are limited to a specific parallel application. It will be interesting to see how well a broad range of parallel applications perform in the HRT model. We plan to explore more parallel applications such as bulk-synchronous parallel applications and applications that rely heavily on message passing. We also plan to explore workloads that are largely memory-bound or I/O-bound.

9. REFERENCES

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. 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)*, pages 95–109, Oct. 1991.

[2] 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.

[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the* 22^{nd} *ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 29–44, Oct. 2009.

[4] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the* 8^{th} *ACM European Conference on Computer Systems (EuroSys 2013)*, pages 239–252, Apr. 2013.

[5] 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* 10^{th} *USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)*, pages 335–348, Oct. 2012.

[6] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw. Data-only flattening for nested data parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013)*, Feb. 2013.

[7] L. Bergstrom and J. Reppy. Nested data-parallelism on the gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, Sept. 2012.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 267–283, Dec. 1995.

[9] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, et al. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, Apr. 1992.

[10] 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.

[11] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the International Conference on Function Programming (ICFP)*, May 1996.

[12] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Apr. 1992.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the* 8^{th} *USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*, pages 43–57, Dec. 2008.

[14] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the* 9^{th} *USENIX Conference on Operating Systems Design and Implementation (OSDI 2010)*, Oct. 2010.

[15] R. Brightwell, R. Oldfield, D. Bernholdt, A. Maccabe, E. Brewer, P. Bridges, P. Dinda, J. Dongarra, C. Iancu, M. Lang, J. Lange, D. Lowenthal, F. Mueller, K. Schwan, T. Sterling, and P. Teller. Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2013)*, June 2013.

[16] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the* 16^{th} *Symposium*

on Operating Systems Principles (SOSP 1997), pages 143–156, Oct. 1997.

[17] S. Cadamb, G. Coviello, C.-H. Li, R. Phull, K. Rao, M. Sankaradass, and S. Chakradhar. Cosmic: Middleware for high performance and reliable multiprocessing on intel manycore coprocessors. In *Proceedings of the 22nd ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2013)*, June 2013.

[18] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[19] 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.

[20] 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.

[21] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel langauge. *International Journal of High Performance Computing Applications*, 21(3):291–312, Aug. 2007.

[22] P. Charles, C. Donawa, K. Ebicioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2005.

[23] J. S. Chase, J. S. Chase, H. M. Levy, H. M. Levy, M. J. Feeley, M. J. Feeley, E. D. Lazowska, and E. D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, Nov. 1994.

[24] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the 1^{st} USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 14:1–14:15, Nov. 1994.

[25] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A multi-platform co-array fortran compiler. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2004.

[26] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5^{th} Workshop on Hot Topics in Operating Systems (HotOS 1995)*, pages 78–83, May 1995.

[27] 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 15^{th} ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 251–266, Dec. 1995.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] S. Hofmeyr, J. A. Colmenares, C. Iancu, and J. Kubiatowicz. Juggle: Proactive load balancing on multicore computers. In *Proceedings of the 20th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2011)*, June 2011.

[34] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, Apr. 2007.

[35] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv—optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014)*, June 2014.

[36] S. Krieder, J. Wozniak, T. Armstrong, M. Wilde, D. Katz, B. Grimmer, I. Foster, and I. Raicu. Design and evaluation of the GeMTC framework for gpu-enabled many-task computing. In *Proceedings of the 23rd ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2014)*, June 2014.

[37] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of the 1^{st} ACM European Conference on Computer Systems (EuroSys 2006)*, pages 133–145, Apr. 2006.

[38] 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.

[39] S. Lee and J. Vetter. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2014)*, June 2014.

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

[41] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović,

and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the* 1^{st} *USENIX Conference on Hot Topics in Parallelism (HotPar 2009)*, pages 10:1–10:6, Mar. 2009.

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. 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)*, pages 461–472, Mar. 2013.

[43] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems (HotOS 1995)*, pages 58–61, May 1995.

[44] NVIDIA Corporation. Dynamic parallelism in CUDA, Dec. 2012.

[45] J. Oayang, B. Kocoloski, J. Lange, and K. Pedretti. Enabling multi-stack software on partitioned hardware for exascale systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2015)*, May 2015.

[46] S. Peter and T. Anderson. Arrakis: A case for the end of the empire. In *Proceedsings of the* 14^{th} *Workshop on Hot Topics in Operating Systems (HotOS 2013)*, pages 26:1–26:7, May 2013.

[47] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. 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)*, pages 291–304, Mar. 2011.

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

[49] 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.

[50] K. B. Wheeler, R. C. Murphy, and D. Thain. 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.

[51] 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.

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