A Case for Transforming Parallel Runtimes 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 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 summarize 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.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

Keywords

hybrid runtimes, HRTs, parallel runtimes, Nautilus

1. INTRODUCTION

Modern parallel runtimes 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 runtime and affect its performance, efficiency, and scalability. First, the runtime is deprived of the use of hardware features that are available only in kernel mode. This is a large portion of the machine. The second consequence is that the runtime must use the abstractions provided by the kernel, even if the abstractions are a bad fit. Finally, the kernel has minimal access to the information available to the parallel runtime or to the language implementation it supports.

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 Xeon 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 runtimes, such as the Legion runtime [1, 22] 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 runtimes such as Legion, and the runtimes 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 give an overview of the case for transforming parallel runtime 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 runtime, the user/kernel abstraction itself, which dates back to Multics, is not a good one. It is 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

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

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749264..

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 runtime have not been first-order concerns.

Runtime 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 runtimes better, a parallel runtime as a user level component is fundamentally *constrained* by the kernel/user abstraction. In contrast, as a kernel, a parallel runtime 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 runtimes.

At first glance, transforming a parallel runtime into a kernel seems to be a particularly daunting task because language runtimes 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 runtime where the bulk of execution time is spent, not on the whole functional base of the runtime. The core of the runtime 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 runtime so the non-core elements run at user-level on top of the full software stack they expect, while the core of the runtime runs as a kernel. We refer to such a kernel as a hybrid runtime (HRT) as it is a hybrid between a kernel and a runtime. Our focus in this paper is on the HRT.

We make the following contributions:

- We describe the limitations of building parallel runtime systems on top of general-purpose operating systems and how these limitations are avoided if the runtime *is* a kernel. That is, we motivate HRTs.
- We introduce Nautilus, a prototype kernel framework designed to facilitate the porting of existing parallel runtimes to run as kernels, as well as the implementation of new parallel runtimes directly as kernels. That is, we create an essential tool for easily making HRTs.
- We summarize our experiences in using Nautilus to transform three runtimes 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.

Readers should refer to our technical report [16]. for a more detailed discussion than is possible in this summary.

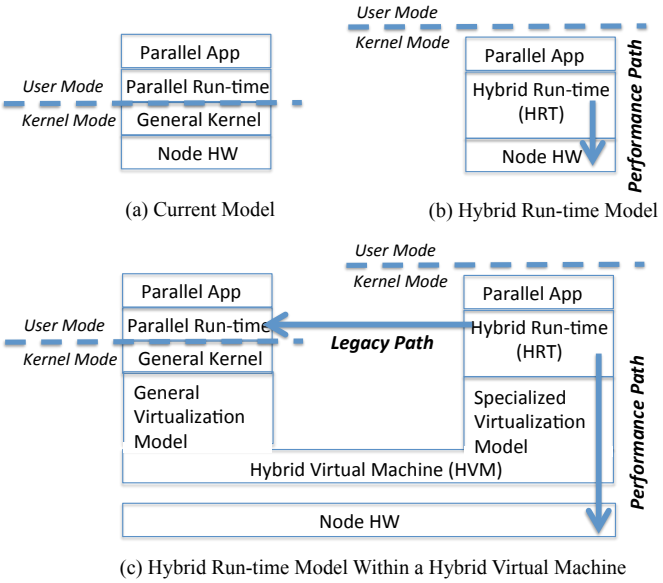

Figure 1: Overview of Hybrid Runtime (HRT) approach: (a) current model used by parallel runtimes, (b) proposed HRT model, and (c) proposed HRT model combined with a hybrid virtual machine (HVM).

2. ARGUMENT

A language's runtime 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 runtime abstracts the properties of both and impedance-matches them 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 runtime 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 runtimes 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 runtime gains more responsibilities and features, the lines between the runtime 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 runtime 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 runtime 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 kernel's abstractions be at odds with the runtime, it may also completely prevent the runtime 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 runtime—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 runtime 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 runtimes, which we call Hybrid Runtimes (HRTs), as they are hybrids of parallel runtimes 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 runtime kernels to have available the full functionality of the general-purpose OS for components not central to runtime operation.

Figure 1 illustrates three different models for supporting a parallel runtime system. The current model (a) layers the parallel runtime over a general-purpose kernel. The parallel runtime runs in user mode without access to privileged hardware features and uses a kernel API designed for general-purpose computations. In the Hybrid Runtime model (b) that we describe in this paper the parallel runtime 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 runtime 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 runtime 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 [18] is currently underway in our group as part of the Hobbes exascale software project [9]. However, it is important to note that other paths exist. For example, Guarded Modules [17] 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 [2] uses hardware virtualization features to provide privileged CPU access to Linux processes. The HVM could be built on top of Dune. The Pisces system [21] 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 runtimes 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 4–5, this allowed us to port three very different runtimes to Nautilus and the HRT model in a very reasonable amount of time. Note that while these initial ports were carried out manually, we are currently investigating how to automate this process.

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. Nautilus currently has been tested on a range of Intel and AMD machines, as well as on the Palacios VMM. A port to the Intel Phi is underway. The machine used for performance evaluation in this paper is a 4 socket, 8 NUMA domain, 64 core server based on 2.1GHz AMD Opteron 6272 (Interlagos) processors and 128 GB of memory.

The design of Nautilus was heavily influenced by early research on microkernels [19, 6, 5] and even more by Engler and others' work on exokernels [12, 13]. Like Nautilus, the exokernel concept involves 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 runtime, but we shed the notion of privilege levels for the sake of functionality and performance.

We stress that the design of Nautilus is, first and foremost, driven by the needs of the parallel runtimes that use it. Nevertheless, it is complete enough to leverage the full capabilities of a modern 64-bit x86 machine to support

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

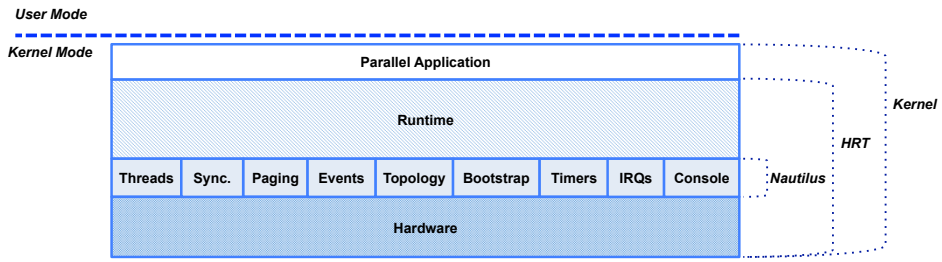

Figure 2: Structure of Nautilus.

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

Figure 3: Source lines of code for the Nautilus kernel.

three runtimes, one of which (Legion) is quite complex and is used in practice today.

Currently, Nautilus is designed to boot the machine, discover its capabilities, devices, and topology, and immediately hand control over to the runtime. 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. A port to the Intel Xeon Phi is currently underway.

We spent a good deal of time measuring the capabilities that affect the performance of the HRTs we built, and were pleased to find that the Nautilus can, in many cases, significantly outperform the only other viable OS for these runtimes, i.e. Linux. For example, Nautilus enjoys a speedup of 30x over Linux for light-weight thread creation. Some of the runtimes we investigated made heavy use of event notification mechanisms. We found that we could provide significantly lighter-weight event notification using hardware features, such as inter-processor interrupts (IPIs), that are not available in user-space. To give an idea of the magnitude of the difference, an event wakeup using an IPI to "kick" the processor waiting for the event produced an 8x speedup over the traditional condition variable implementation.

3.1 Complexity

We now make a case for the potential for Nautilus as a vehicle for HRTs, setting aside the attractive raw performance of its primitives and focusing on implementation complexity.

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 3 shows that Nautilus is fairly compact at ∼24,000 lines of code.

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, discussed at length in the next section, was the most challenging and complex of the three runtimes 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 runtimes took a total of about four person-months of effort. Most of this work went into understanding Legion and its needs. Only about 800 lines of code needed to be added to Nautilus to support Legion and the other two runtimes. This is tiny considering the size of the Legion runtime and the others.

This suggests that exploring the HRT model for existing or new parallel runtimes, 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 runtime system is designed to provide applications with a parallel programming model that maps well to heterogeneous architectures [1, 22]. 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. 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 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 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 runtime.

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 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 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 *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. As we are interested in single-node performance, we naturally focused our efforts on the shared low-level Legion runtime. 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 runtime, the port was fairly straight-forward.

Legion, in its default user-level implementation, uses pthreads as representations of logical processors, so the low-level runtime 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. After porting Legion into Nautilus, we then began to explore how Legion could benefit from unrestricted access to the machine.

We now present a brief evaluation of our transformation of the user-level Legion runtime 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 4, 2014, specifically, commit e22962d, which can be found via the Legion project web site.[2]

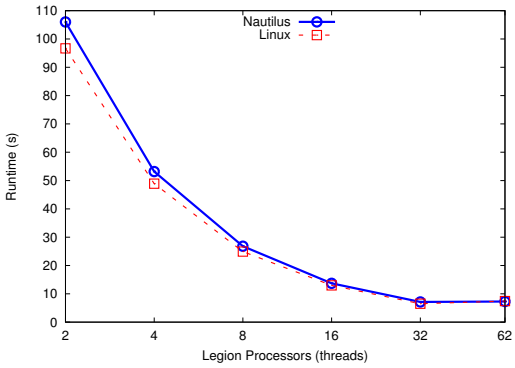

Figure 4: Run time of Legion circuit simulator versus core count.

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 to execute Legion tasks. Figure 4 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 livelock 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.

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

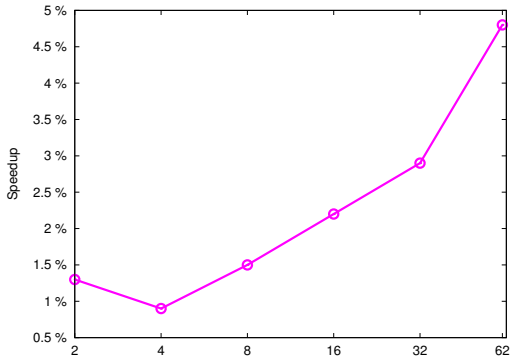

Figure 5: Speedup of Legion circuit simulator comparing the baseline Nautilus version and a Nautilus version that executes Legion tasks with interrupts disabled.

To experiment with hardware functionality in the HRT model, we wanted to take advantage of a capability that normally is not available in Linux at user-level. We decided to use the capability to disable interrupts. In the Legion HRT, there 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 5 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 runtime performance.

5. EXAMPLE: NESL AND NDPC

NESL [7] 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 [14, 15], Data Parallel Haskell [10, 11], and arguably the nested call extensions to CUDA [20]. 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 [8] and very recent work on using NESL-like nested data parallelism for

GPUs [4] and multicore [3] 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, pointwise vector operations execute in parallel.

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. It ties specifically to a fast thread fork mechanism implemented in Nautilus.

We believe that our transformations of NESL and NDPC into the HRT model show that adapting a parallel language to this model does not require a monumental effort. This provides an opportunity for parallel runtime developers to quickly prototype their runtime implementation with unrestricted access to hardware using Nautilus.

6. CONCLUSIONS

We have summarized the case for transforming parallel runtimes into operating system kernels, forming hybrid runtimes (HRTs). The motivations for HRTs include the increasing complexity of hardware, the convergence of parallel runtime concerns and abstractions in managing such hardware, and the limitations of executing the runtime at user-level, both in terms of limited hardware access and limited control over kernel abstractions. We introduced Nautilus, a prototype kernel framework to facilitate the construction of HRTs. Using Nautilus, we were able to successfully transform three very different runtimes into HRTs. For the Legion runtime, 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 runtimes was a four person-month effort. It is somewhat remarkable that even with a fairly nascent kernel framework, *just* by dropping the runtime down to kernel level and taking advantage of a kernel-only feature in two lines of code, we can exceed performance on Linux, which has undergone far more substantial development and tuning.

7. REFERENCES

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

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

[3] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw. 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)*, pages 81–92, Feb. 2013.

[4] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. In *Proceedings of the* 17^{th} *ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 247–258, Sept. 2012.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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* 3^{rd} *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2013)*, June 2013.

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

[11] 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.

[12] 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.

[13] 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.

[14] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status report: The manticore project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*, October 2007.

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

[16] K. C. Hale and P. Dinda. Details of the case for transforming parallel runtimes into operating system kernels. Technical Report NU-EECS-15-01, Department of Computer Science, Northwestern University, Apr. 2015.

[17] 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.

[18] 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* 24^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Apr. 2010.

[19] 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.

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

[21] J. Ouyang, B. Kocoloski, J. Lange, and K. Pedretti. 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.

[22] 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.