Automatic Hybridization of Runtime Systems

Kyle C. Hale Conor J. Hetland Peter A. Dinda
{kh,ch}@u.northwestern.edu, pdinda@northwestern.edu
Department of Electrical Engineering and Computer Science
Northwestern University

ABSTRACT

The hybrid runtime (HRT) model offers a plausible path towards high performance and efficiency. By integrating the OS kernel, parallel runtime, and application, an HRT allows the runtime developer to leverage the full privileged feature set of the hardware and specialize OS services to the runtime's needs. However, conforming to the HRT model currently requires a *complete* port of the runtime and application to the kernel level, for example to our Nautilus kernel framework, and this requires knowledge of kernel internals. In response, we developed Multiverse, a system that bridges the gap between a built-from-scratch HRT and a legacy runtime system. Multiverse allows existing, unmodified applications and runtimes to be brought into the HRT model without any porting effort whatsoever. Developers simply recompile their package with our compiler toolchain, and Multiverse automatically splits the execution of the application between the domains of a legacy OS and an HRT environment. To the user, the package appears to run as usual on Linux, but the bulk of it now runs as a kernel. The developer can then *incrementally* extend the runtime and application to take advantage of the HRT model. We describe the design and implementation of Multiverse, and illustrate its capabilities using the Racket runtime system.

1. INTRODUCTION

Runtime systems can gain significant benefits from executing in a tailored software environment. In previous work, we proposed one such specialized environment called the Hybrid Runtime (HRT) [10, 11]. In an HRT, a light-weight kernel framework (called an AeroKernel), a runtime, and an application coalesce into a single entity in which the runtime can enjoy full access to the underlying hardware, including features typically reserved for a privileged OS.

An AeroKernel can export functionality through a standard interface such as POSIX or through a custom interface. However, it exists solely for convenience, and the runtime may not even leverage the mechanisms it provides. Ultimately, the choices of proper execution model and abstractions to the hardware are left to the runtime. The runtime developers can build or choose the kernel abstractions they need. The motivation for an AeroKernel draws from the reliable performance of light-weight kernels [16, 15, 8], the philosophy regarding kernel abstractions of Exokernel [4], new techniques and ideas developed in multi-core OS research [17, 5], and the simplicity of other experimental OSes from previous decades [14, 18]. In this paper, we use our Nautilus AeroKernel, which we describe in more detail in Section 2.

Prior to the work and system we describe here, the implementation of an HRT consisted entirely of manual processes. HRT developers needed first to extend an AeroKernel framework such as Nautilus with the functionality the runtime needed. The HRT developers would then port the runtime to this AeroKernel manually. Readers interested in finer detail regarding this process can refer to our technical report [11]. While a manual port can produce the highest performance gains, it requires an intimate familiarity with the runtime system's functional requirements, which may not be obvious. These requirements must then be implemented in the AeroKernel layer and the AeroKernel and runtime must be combined. This requires a deep understanding of kernel development. This manual process is also iterative: the developer adds AeroKernel functionality until the runtime works correctly. The end result might be that the AeroKernel interfaces support a small subset of POSIX, or that the runtime developer replaces such functionality with custom interfaces.

While such a development model *is* tractable, and we have transformed three runtimes to HRTs using it, it represents a substantial barrier to entry to creating HRTs, which we seek here to lower. The manual porting method is *additive* in its nature. We must add functionality until we arrive at a working system. A more expedient method would allow us to *start* with a working HRT produced by an automatic process, and then incrementally extend it and specialize it to enhance its performance.

The Multiverse system we describe in this paper supports

This project is made possible by support from the United States National Science Foundation through grant CCF-1533560 and from Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the United States Department of Energy's Office of Science.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907309

just such a method using a technique called *automatic hybridization* to create a working HRT from an existing, unmodified runtime and application. With Multiverse, runtime developers can take an incremental path towards adapting their systems to run in the HRT model. From the user's perspective, a hybridized runtime and application behaves the same as the original. It can be run from a Linux command line and interact with the user just like any other executable. But internally, it executes in kernel mode as an HRT.

Multiverse bridges a specialized HRT with a legacy environment by borrowing functionality from a legacy OS, such as Linux. Functions not provided by the existing AeroKernel are forwarded to another core that is running the legacy OS, which handles them and returns their results. The runtime developer can then identify hot spots in the legacy interface and move their implementations (possibly even changing their interfaces) into the AeroKernel. The porting process with Multiverse is *subtractive* in that a developer iteratively removes dependencies on the legacy OS. At the same time, the developer can take advantage of the kernel-level environment of the HRT.

To demonstrate the capabilities of Multiverse, we automatically hybridize the Racket runtime system. Hybridized Racket executes in kernel mode as an HRT, and yet the user sees precisely the same interface (an interactive REPL environment, for example) as out-of-the-box Racket.

Our contributions in this paper are as follows:

- We introduce the concept of *automatic hybridization* for transforming runtime systems and their applications into HRTs, enabling them to run in kernel mode with full access to hardware features and the ability to adapt the kernel to their needs.
- We describe the design of Multiverse, an implementation of automatic hybridization that combines compile-time, link-time, run-time, and virtualization-based techniques.
- We demonstrate automatic hybridization with Multiverse by transforming the Racket runtime into an HRT.
- We evaluate the initial performance of Multiverse.

2. HRT AND HVM

Multiverse builds on our previously described work and systems [10, 11, 9] to define and support the hybrid runtime (HRT) model. We describe the key salient findings and components here.

The core premise of the HRT model is that by moving the parallel runtime (and its application) to the kernel level, we enable the runtime developer to leverage all hardware features (including privileged features), and to specialize kernel features specifically for the runtime's needs. These capabilities in turn allow for greater performance or efficiency than is possible at user-level. We developed a kernel framework, the Nautilus AeroKernel, to facilitate doing this. Nautilus runs on bare metal or under virtualization on x64 machines and the Intel Xeon Phi.

We have previously ported three runtimes to Nautilus, namely Legion [2], the NESL VCODE interpreter [3], and the runtime of a home-grown nested data parallel language. Using the HPCG (High Performance Conjugate Gradients) benchmark [13] developed by Sandia National Labs and ported to Legion by Los Alamos National Labs, we demonstrated speedups over Linux of up to 20% for the Intel Xeon Phi, and up to 40% for a 4-socket, 64-core x64 AMD Opteron 6272 machine.

Multiverse also builds on the Hybrid Virtual Machine (HVM), an extension to the open source (BSD) Palacios VMM [16]. HVM allows for the creation of a VM whose memory, cores, and interrupt logic are segregated so that one VM simultaneously runs two operating systems, the "Regular Operating System" (ROS) (e.g., Linux) and an HRT-based OS (e.g., Nautilus). The ROS runs on a partition of the cores and can see and touch only the ROS cores and the ROS subset of physical memory. In contrast, the HRT, while only allowed to run on its own distinct partition of the cores, has full access to all the memory, cores, and interrupt logic of the entire VM. The ROS and HRT can be booted and rebooted independently.

3. MULTIVERSE

The goal of Multiverse is to ease the path for developers of transforming a runtime into an HRT. We seek to make the system look like a compilation toolchain option from the developer's perspective. That is, to the greatest extent possible, the HRT is a compilation target. Compiling to an HRT results in an executable that is a "fat binary" containing additional code and data that enables kernel-mode execution in an environment that supports it. An HVM-enabled VM on Palacios is the first such environment. The developer can extend this incrementally—Multiverse facilitates a path for runtime and application developers to explore how to specialize their HRT to the full hardware feature set and the extensible kernel environment of the AeroKernel.

From the user's perspective, the executable behaves as if it were compiled for a standard user-level Linux environment. The user sees no difference between HRT execution and user-level execution.

3.1 Techniques

The Multiverse system relies on three key techniques: state split execution, event channels, and state superpositions. We now describe each of these.

Split execution. In Multiverse, a runtime and its application begin their execution in the ROS. Through a well-defined interface, the runtime on the ROS side can spawn an execution context in the HRT. At this point, Multiverse splits its execution into two components, each running in a different context; one executes in the ROS and the other in the HRT. The semantics of these execution contexts differ from traditional threads depending on their characteristics. In the current implementation, the context on the ROS side comprises a Linux thread, the context on the HRT side comprises an AeroKernel thread, and we refer to them collectively as an *execution group*. While execution groups in our current system consist of threads in different OSes, this need not be true in general. The context on the HRT side executes until it triggers a fault, a system call, or other event. The execution group then converges on this event, with each side participating in a protocol for requesting events and receiving results. This protocol exchange occurs in the context of HVM event channels, which we discuss below.

Figure 1 illustrates the split execution of Multiverse for a ROS/HRT execution group. At this point, the ROS has already made a request to create a new context in the HRT. When the HRT thread begins executing in the HRT side, exceptional events, such as page faults, system calls, and other exceptions vector to stub handlers in the AeroKernel

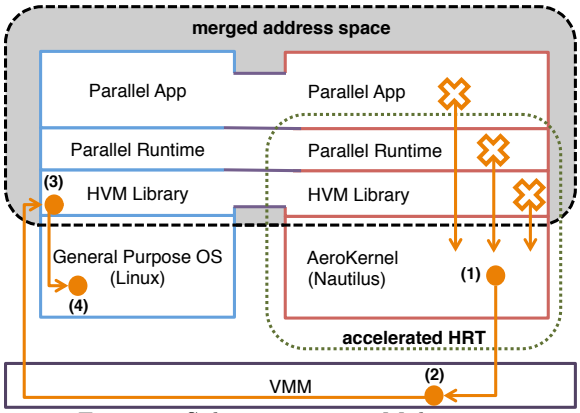

Figure 1: Split execution in Multiverse.

Item	Cycles	Time
Address Space Merger	∼33 K	1.5 μs
Asynchronous Call	∼25 K	1.1 μs
Synchronous Call (different socket)	∼1060	48 ns
Synchronous Call (same socket)	∼790	36 ns

Figure 2: Round-trip latencies of ROS↔HRT interactions.

(1). The AeroKernel then redirects these events through an event channel (2) to request handling in the ROS. The VMM then injects these into the originating ROS thread, which can take action on them directly (3). For example, in the case of a page fault that occurs in the ROS portion of the virtual address space, the HVM library simply replicates the access, which will cause the same exception to occur on the ROS core. The ROS will then handle it as it would normally. In the case of events that need direct handling by the ROS kernel, such as system calls, the HVM library can simply forward them (4).

Event channels. When the HRT needs functionality that the ROS implements, access to that functionality occurs over *event channels*, event-based, VMM-controlled communication channels between the two contexts. The VMM only expects that the execution group adheres to a strict protocol for event requests and completion.

Figure 2 shows the measured latency of event channels with the Nautilus AeroKernel performing the role of HRT. The first two calls are bounded from below by the latency of hypercalls to the VMM, while the remainder operate at memory synchronization speeds.

State superpositions. In order to forego the addition of burdensome complexity to the AeroKernel environment, it helps to leverage functions in the ROS other than those that lie at a system call boundary. This includes functionality implemented in libraries and more opaque functionality like optimized system calls in the `vdso` and the `vsyscall` page. In order to use this functionality, Multiverse can set up the HRT and ROS to share portions of their address space, in this case the user-space portion. Aside from the address space merger itself, Multiverse leverages other state superpositions to support a shared address space, including superpositions of the ROS GDT and thread-local storage state.

In principle, we could superimpose any piece of state visible to the VMM. The ROS or the runtime need not be aware of this state, but the state is nonetheless necessary for facilitating a simple and approachable usage model.

The superposition we leverage most in Multiverse is a merged address space between the ROS and the HRT. The merged address space allows execution in the HRT without a need for implementing ROS-compatible functionality. When a merged address space takes effect, the HRT can use the same user-mode virtual addresses present in the ROS. For example, the parallel runtime in the ROS might load files and construct a complex pointer-based data structure in memory. It can then invoke a function within its counterpart in the HRT to compute over that data.

4. EVALUATION

In this section we evaluate Multiverse using a hybridized Racket runtime system running a set of benchmarks from The Language Benchmark Game. We ran all experiments on a Dell PowerEdge 415 with 8GB of RAM and an 8 Core 64-bit x86_64 AMD Opteron 4122 clock clocked at 2.2GHz. Each CPU core has a single thread with four cores per socket. The host machine has stock Fedora Linux 2.6.38.6-26.rc1.fc15.x86_64 installed, and is configured for maximum performance in the BIOS. Benchmark results are reported as averages of 10 runs.

Experiments in a VM were run on a guest setup which consists of a simple BusyBox distribution running an unmodified Linux 2.6.38-rc5+ image with two cores (one core for the HVM and one core for the ROS) and 1 GB of RAM.

Racket

Racket [7, 6] is the most widely used Scheme implementation and has been under continuous development for over 20 years. It is an open source codebase that is downloaded over 300 times per day. Recently, support has been added to Racket for parallelism via futures [19] and places [20].

The Racket runtime is a good candidate to test Multiverse, particularly its most complex usage model, the incremental model, because Racket includes many of the challenging features emblematic of modern dynamic programming languages that make extensive use of the Linux ABI, including system calls, memory mapping, processes, threads, and signals. Readers can find more details on the various usage models of Multiverse in our technical report [12].

To evaluate the correctness and performance of our port, we tested it on a series of benchmarks submitted to The Computer Language Benchmarks Game [1]. We tested on seven different benchmarks: a garbage collection benchmark (binary-tree-2), a permutation benchmark (fannkuch), two implementations of a random DNA sequence generator (fasta and fasta-3), a generation of the mandelbrot set (mandelbrot-2), an n-body simulation (n-body), and a spectral norm algorithm.

Note that while this is an implementation of a high-level language, the actual execution of Racket programs involves many interactions with the operating system. These exercise Multiverse's system call and fault forwarding mechanisms.

Figure 3 compares the performance of the Racket benchmarks run natively on our hardware, under virtualization, and as an HRT that was created with the initial implementation of Multiverse. The overhead of the Multiverse case compared to the virtualized and native cases is due to the frequent interactions with the Linux ABI. Most of these in-

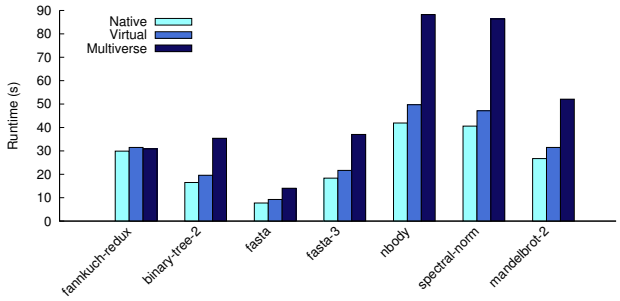

Figure 3: Performance of Racket benchmarks running Native, Virtual, and in Multiverse. Note that the Multiverse result is the result of Multiverse's automatic hybridization of Racket—it is the *starting point* for incremental enhancement within the HRT model.

teractions arise from page faults rather than system calls. In the Multiverse case, these are forwarded from the HRT to the ROS to be handled.

It is worth reflecting on what exactly has happened here: we have taken a complex runtime system off-the-shelf, run it through Multiverse without changes, and as a result have a version of the runtime system that correctly runs in kernel mode as an HRT and behaves identically with virtually identical performance. To be clear, **all of the Racket runtime except Linux kernel ABI interactions is seamlessly running as a kernel.** While this codebase is the endpoint for user-level development, it represents a *starting point* for HRT development in the incremental model.

5. CONCLUSIONS AND FUTURE WORK

We introduced Multiverse, a system that implements *automatic hybridization* of runtime systems in order to transform them into hybrid runtimes (HRTs). We illustrated the design and implementation of Multiverse and described how runtime developers can use it as a tool for incremental porting of runtimes and applications from a legacy OS to a specialized AeroKernel.

To demonstrate its power, we used Multiverse to automatically hybridize the Racket runtime system, a complex, widely-used, JIT-based runtime. With automatic hybridization, we can take an existing Linux version of a runtime or application and automatically transform it into a package that looks to the user as if it runs like any other program, but actually executes on a remote core in kernel-mode, in the context of an HRT, and with full access to the underlying hardware. We evaluated the performance overheads of an unoptimized Multiverse hybridization of Racket and showed that performance varies with the usage of legacy functionality. Runtime developers can leverage Multiverse to start with a working system and incrementally transition heavily utilized legacy functions to custom components within an AeroKernel.

6. REFERENCES

[1] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

[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] 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.

[4] 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.

[5] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec. 1992.

[6] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. The Racket Manifesto. In T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[8] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of Supercomputing (SC 2010)*, Nov. 2010.

[9] K. Hale and P. Dinda. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, April 2016.

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

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

[12] K. C. Hale, C. Hetland, and P. A. Dinda. Automatic hybridization of runtime systems. Technical Report NWU-EECS-16-03, Department of Computer Science, Northwestern University, Mar. 2016.

[13] M. A. Heroux, J. Dongarra, and P. Luszczek. HPCG technical specification. Technical Report SAND2013-8752, Sandia National Laboratories, October 2013.

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

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

[16] 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.

[17] 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.

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

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

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