NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
Number: NU-EECS-16-03

March, 2016

Automatic Hybridization of Runtime Systems

Kyle C. Hale, Conor Hetland, and Peter Dinda

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.

Keywords: Multiverse, runtime systems, Hybrid Runtimes, operating systems

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.

Automatic Hybridization of Runtime Systems

Kyle C. Hale Conor 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) [19, 20]. 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 to runtimes 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 [25, 23, 17], the philosophy regarding kernel abstractions of Exokernel [13], new techniques and ideas developed in multi-core OS research [26, 14], and the simplicity of other experimental OSes from previous decades [22, 33]. 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 [20]. 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 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 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 de-

veloper 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. Racket has a complex, JIT-based runtime system with garbage collection and makes extensive use of the Linux system call interface, memory protection mechanisms, and external libraries. 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 outline three usage models (native, accelerator, and incremental) supported by Multiverse, as well as AeroKernel overrides, a mechanism by which developers can override legacy functionality with high-performance variants.
- We demonstrate automatic hybridization with Multiverse by transforming the Racket runtime into an HRT.
- We evaluate the performance of Multiverse.

Multiverse, and the implementations of the techniques and underlying technologies describe here either are or will be publicly available within the open source codebases of the Nautilus AeroKernel and the Palacios VMM.

2. HRT AND HVM

Multiverse builds on our previously described work and systems [19, 20, 18] 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 possible at user-level. We have developed a kernel framework, named the Nautilus AeroKernel, to facilitate doing exactly this. Nautilus runs on bare metal or under virtualization on x64 machines and the Intel Xeon Phi. It is open source (MIT) and its repository is accessible from our web site.

We have previously hand-ported three runtimes to Nautilus, namely Legion [4], the NESL VCODE interpreter [9], and the runtime of a home-grown nested data parallel language. Using the HPCG (High Performance Conjugate Gradients) benchmark [11, 21] 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. Nautilus provides basic primitives, for example thread creation and events, that outperform Linux by orders of magnitude because we designed them to support runtimes in lieu of general-purpose computing, and because there are no kernel/user boundaries to cross. This combined with hardware and software capabilities available only in kernel mode, for example complete interrupt control and runtime-specific scheduling, leads to these performance gains in applications.

Multiverse also builds on the Hybrid Virtual Machine (HVM), an extension to the open source (BSD) Palacios VMM [25], available in its repository, and also accessible from our web site. 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 sees and can 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.

Because in Palacios, as in most VMMs, the storage of virtualization state and handling of VM exit events are carried out on a per-virtual core basis, the segregation of ROS cores and HRT cores allows for distinct policies to be applied to the ROS and the HRT, and for each to have distinct features. We use this fact to make the virtualization layer for the HRT much "thinner" than for the ROS. For example, because the HVM and HRT collaborate on establishing a simple address space, exits for page faults in either nested or shadow paging are rare. Because typical I/O devices are handled by the ROS, the HRT never sees their interrupts (or exits relating to their underlying physical devices). The HRT can even avoid exits attributable to timer interrupts if it does not use a timing device. Finally, Palacios's existing memory management allows for the physical memory underlying the HRT to be mapped to appropriate NUMA zones that can be segregated from those used by the ROS. A rudimentary cache partitioning mechanism also exists if it is necessary to share the last-level cache in a socket. For this reason, during normal operation and when running out of HRT-only memory, a Nautilus-based HRT can achieve essentially the same performance as it would natively.

The HVM portion of Palacios provides hypercall and shared memory-based mechanisms for the ROS and HRT to communicate and for both to communicate to the HVM. HVM supports both asynchronous and synchronous communication models, as well as signaling. On the previously mentioned x64 hardware, asynchronous communication latency and signaling latency is about 11 μs, while synchronous communication latency is 359-482 ns depending on the distance between physical cores in the machine structure.

On the ROS side, the communication endpoint is the user application, not the ROS kernel. That is, the HRT acts as an extension of the application, which is important for Multiverse, as we describe below. Due to a specialized boot protocol, an extension of the multiboot2 standard, the HRT can be booted or rebooted in just milliseconds, putting HRT boot at a cost on par with a process `fork()+exec()` in the ROS. The HVM allows a user-level ROS application to sup-

ply the HRT image via a hypercall, much like an `exec()`.

Asynchronous HRT-to-ROS signaling bypasses the ROS kernel. The ROS application registers a signal handler function and stack with the HVM, similar to how the canonical `signal()` library function is used. When the HRT raises a signal, the HVM records the signal raise and begins to watch for an opportunity to inject it. When handling an entry is about to occur in user-mode context, and with CR3 set to the registering process's CR3 value, the HVM builds what looks like an interrupt frame on the supplied stack, and then reenters the guest with the instruction pointer pointing at the supplied handler and stack pointer pointing at the supplied stack. In effect, this is an "interrupt to user" construct which is lower priority than any actual exception or interrupt, and lower priority than the guest's own signals. In contrast, asynchronous ROS-to-HRT signaling occurs via exception injection and hence signaling from the ROS application takes highest precedence within the HRT.

3. MULTIVERSE

We designed the Multiverse system to support automatic hybridization of existing runtimes and applications that run in user-level on Linux platforms.

3.1 Perspectives

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 simply 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 virtual machine on Palacios is the first such environment. The developer can then 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 just as if it were compiled for a standard user-level Linux environment. The user sees no difference between HRT execution and user-level execution.

3.2 Techniques

The Multiverse system relies on three key techniques: state superpositions, split execution, and event channels. 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 discussed in Section 3.3, 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. We discuss these differences in Section 4. 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.

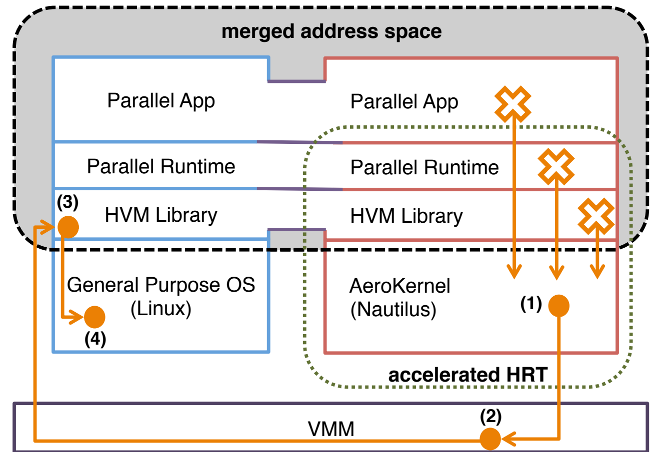

Figure 1: Split execution in Multiverse.

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.

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, e.g. through an asynchronous function invocation. 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 (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. Note that these calls are bounded from below by the latency

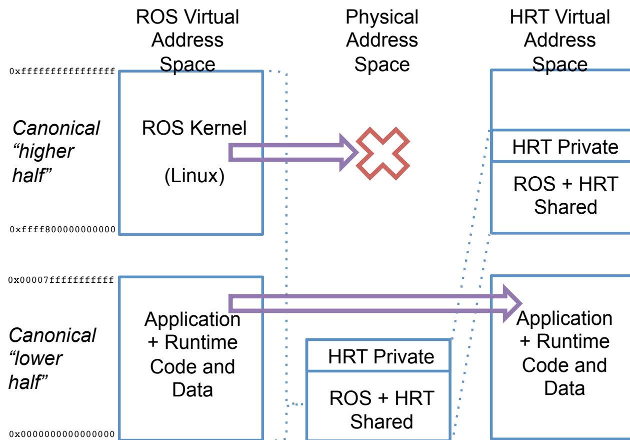

Figure 3: Merged address space between ROS and HRT.

of hypercalls to the VMM.

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, depicted in Figure 3. 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.

3.3 Usage models

The Multiverse system is designed to give maximum flexibility to application and runtime developers in order to encourage exploration of the HRT model. While the degree to which a developer leverages Multiverse can vary, for the purposes of this paper we classify the usage model into three categories, discussed below.

Native In the native model, the application/runtime is ported to operate fully within the HRT/AeroKernel setting. That is, it does not use any functionality not exported by the AeroKernel, such as `glibc` functionality or system calls like `mmap()`. This category allows maximum performance, but requires more effort, especially in the compilation process. The ROS side is essentially unnecessary for this usage model, but may be used to simplify the initiation of HRT execution (e.g. requesting an HRT boot). The native model is also native in another sense: it can execute on bare metal without any virtualization support.

Accelerator In this category, the app/runtime developer leverages both legacy (e.g. Linux) functionality and AeroKernel functionality. This requires less effort, but allows the developer to explore some of the benefits of running their code in an HRT. Linux functionality is enabled by the merged address space discussed previously, but the developer can also leverage AeroKernel functions.

```
static void*
routine (void * in) {
    void * ret = aerokernel_func();
    printf("Result = %d\n", ret);
}

int main (int argc, char ** argv) {
    hrt_invoke_func(routine);
    return 0;
}
```

Figure 4: Example of user code adhering to the accelerator model.

Figure 4 shows a small example of code that will create a new HRT thread and use event channels and state superposition to execute to completion. Runtime initialization is opaque to the user, much like C runtime initialization code. When the program invokes the `hrt_invoke_func()` call, the Multiverse runtime will make a request to the HVM to run `routine()` in a new thread on the HRT core. Notice how this new thread can call an AeroKernel function directly, and then use the standard `printf()` routine to print its result. This `printf` call relies both on a state superposition (merged address space) for the function call linkage to be valid, and on event channels, which will be used when the C library code invokes a system call (e.g. `write()`).

Incremental The application/runtime executes in the HRT context, but does not leverage AeroKernel functionality. Benefits are limited to aspects of the HRT *environment*. However, the developer need only recompile their application to explore this model. Instead of raising an explicit HRT thread creation request, Multiverse will create a new thread in the HRT corresponding to the program's `main()` routine. The Incremental model also allows parallelism, as legacy threading functionality automatically maps to the corresponding AeroKernel functionality with semantics matching those used in pthreads. The developer can then incrementally expand their usage of hardware- and AeroKernel-specific features.

While the accelerator and incremental usage models rely on the HVM virtualized environment of Palacios, it is important to note that they could also be built on physical partitioning [29] as well. At its core, HVM provides to Multiverse a resource partitioning, the ability to boot multiple kernels simultaneously on distinct partitions, and the ability for these kernels to share memory and communicate.

3.4 AeroKernel Overrides

One way a developer can enhance a generated HRT is through *function overrides*. The AeroKernel can implement

functionality that conforms to the interface of, for example, a standard library function, but that may be more efficient or better suited to the HRT environment. This technique allows users to get some of the benefits of the accelerator model without any explicit porting effort. However, it is up to the AeroKernel developer to ensure that the interface semantics and any usage of global data make sense when using these function overrides. Function overrides are specified in a simple configuration file that is discussed in Section 4.

```
static void*
routine (void * in) {
    void * ret = aerokernel_func();
    printf("Result = %d\n", ret);
}

int main (int argc, char ** argv) {
    pthread_t t;
    pthread_create(&t, NULL, routine, NULL
        );
    pthread_join(t, NULL);
    return 0;
}
```

Figure 5: Example of user code adhering to the accelerator model with overrides.

Figure 5 shows the same code from Figure 4 but using function overrides. Here the AeroKernel developer has overridden the standard pthreads routines so that `pthread_create()` will create a new HRT thread in the same way that `hrt_invoke_func()` did in the previous example.

3.5 Toolchain

The Multiverse toolchain consists of two main components, the runtime system code and the build setup. The build setup consists of build tools, configuration files, and an AeroKernel binary provided by the AeroKernel developer. To leverage Multiverse, a user must simply integrate their application or runtime with the provided Makefile and rebuild it. This will compile the AeroKernel components necessary for HRT operation and the Multiverse runtime system, which includes function overrides, AeroKernel binary parsing routines, exit and signal handlers, and initialization code, into the user program.

4. IMPLEMENTATION

We now discuss implementation details for the runtime components of the Multiverse system. This includes the portion of Multiverse that is automatically compiled and linked into the application's address space at build time and the parts of Nautilus and the HVM that support event channels and state superpositions. Unless otherwise stated, we assume the Incremental usage model discussed in Section 3.3.

4.1 Multiverse runtime initialization

As mentioned in Section 3, a new HRT thread must be created from the ROS side (the originating ROS thread). This, however, requires that an AeroKernel be present on the requested core to create that thread. The runtime component (which includes the user-level HVM library) is in charge of booting an AeroKernel on all required HRT cores during program startup. They can either be booted on demand or at application startup. We use the latter in our current setup.

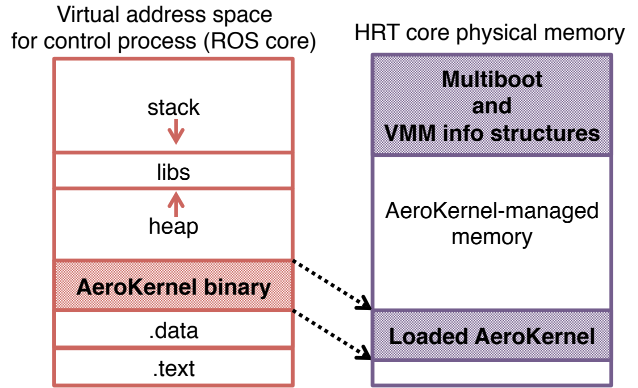

Figure 6: AeroKernel boot process.

Our toolchain inserts program initialization hooks before the program's `main()` function, which carry out runtime initialization.

Initialization tasks include the following:

- Registering ROS signal handlers
- Hooking process exit for HRT shutdown
- AeroKernel function linkage
- AeroKernel image installation in the HRT
- AeroKernel boot
- Merging ROS and HRT address spaces

AeroKernel Boot Our toolchain embeds an AeroKernel binary into the ROS program's ELF binary. This is the image to be installed in the HRT. At program startup, the Multiverse runtime component parses this embedded AeroKernel binary and sends a request to the HVM asking that it be installed in physical memory, as shown in Figure 6. Multiverse then requests the AeroKernel be booted on that core. The boot process, which we described in detail previously [20], brings the AeroKernel up into an event loop that waits for HRT thread creation requests.

The above initialization tasks are opaque to the user, who needs (in the Accelerator usage model) only understand the interfaces to create execution contexts within the HRT.

4.2 Execution model

To implement split execution, we rely on HVM's ability to forward requests from the ROS core to the HRT, along with event channels and merged address spaces.

The runtime developer can leverage two mechanisms to create HRT threads, as discussed in Section 3.3. Furthermore, two types of threads are possible on the HRT side: top-level threads and nested threads. Top-level threads are threads that the ROS explicitly creates. A top-level HRT thread can create its own child threads as well; we classify these as nested threads. The semantics of the two thread types differ slightly in their operation. Nested threads resemble pure AeroKernel threads, but their execution can proceed in the context of the ROS user address space. Top-level threads require extra semantics in the HRT and in the Multiverse component linked with the ROS application.

Threads: Multiverse pairs each top-level HRT thread with a *partner* thread that executes in the ROS. The purpose of this thread is two-fold. First, it allows us to preserve

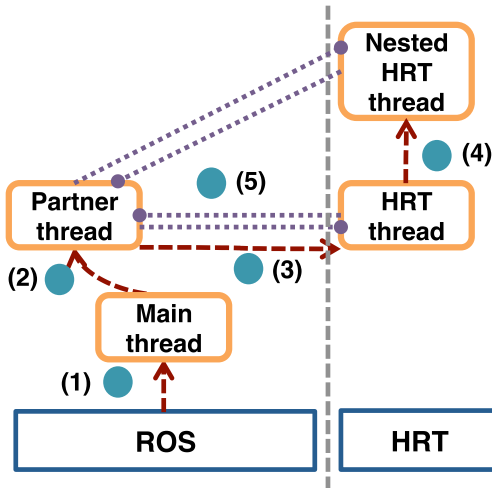

Figure 7: Interactions within an execution group.

join semantics. Second, it gives us the proper thread context in the ROS to initiate a state superposition for the HRT. Figure 7 depicts the creation of HRT threads and their interaction with the ROS. First, in (1) the main thread is created in the ROS. This thread sets up the runtime environment for Multiverse. When the runtime system creates a thread, e.g. with `pthread_create()` or with `hrt_invoke_func()`, Multiverse creates a corresponding partner thread that executes in the ROS (2). It is the duty of the partner thread to allocate a ROS-side stack for a new HRT thread then invoke the HVM to request a thread creation in the HRT using that stack (3). When the partner creates the HRT thread, it also sends over information to initiate a state superposition that mirrors the ROS-side GDT and ROS-side architectural state corresponding to thread-local storage (primarily the `%fs` register). The HRT thread can then create as many nested HRT threads as it desires (4). Both top-level HRT threads and nested HRT threads raise events to the ROS through event channels with the top-level HRT thread's corresponding partner acting as the communication end-point (5).

As is typical in threading models, the main thread can wait for HRT threads to finish by using `join()` semantics, where the joining thread blocks until the child exits. While in theory we could implement the ability to join an HRT thread directly, it would add complexity to both the HRT and the ROS component of Multiverse. Instead, we chose to allow the main thread to join a partner thread directly and provide the guarantee that a partner thread will not exit until its corresponding HRT thread exits on the remote core. When an HRT thread exits, it signals the ROS of the exit event. When Multiverse creates an HRT thread, it keeps track of the Nautilus thread data (sent from the remote core after creation succeeds), which it uses to build a mapping from HRT threads to partner threads. The thread exit signal handler in the ROS flips a bit in the appropriate partner thread's data structure notifying it of the HRT thread completion. The partner can then initiate its cleanup routines and exit, at which point the main thread will be unblocked from its initial `join()`.

Disallowed functionality: Because of the limitations of our current AeroKernel implementation, we must prohibit the ROS code executing in HRT context from leveraging certain functionality. This includes calls that create new execution contexts or rely on the Linux execution model such as `execve`, `clone`, and `futex`. This functionality could, of course, be provided in the AeroKernel, but we have not implemented it at the time of this writing.

Function overrides: In Section 3.3 we described how a developer can use function overrides to select AeroKernel functionality over default ROS functionality. The Multiverse runtime component enforces default overrides that interpose on `pthread` function calls. All function overrides operate using function wrappers. For simple function wrappers, the AeroKernel developer can simply make an addition to a configuration file included in the Multiverse toolchain that specifies the function's attributes and argument mappings between the legacy function and the AeroKernel variant. This configuration file then allows Multiverse to automatically generates function wrappers at build time.

When an overridden function is invoked, the wrapper runs instead, consults a stored mapping to find the symbol name for the AeroKernel variant, and does a symbol lookup to find its HRT virtual address. This symbol lookup currently occurs on every function invocation, so incurs a non-trivial overhead. A symbol cache, much like that used in the ELF standard, could easily be added to improve lookup times. When the address of the AeroKernel override is resolved, the wrapper then invokes the function directly (since it is already executing in the HRT context where it has appropriate page table mappings for AeroKernel addresses).

4.3 Event channels

The HVM model enables the building of essentially any communication mechanism between two contexts (in our case, the ROS and HRT), and most of these require no specific support in the HVM. As a consequence, we minimally define the *basic* communication between the ROS, HRT, and the VMM using shared physical memory, hypercalls, and interrupts.

The user-level code in the ROS can use hypercalls to sequentially request HRT reboots, address space mergers (state superpositions), and asynchronous sequential or parallel function calls. The VMM handles reboots internally, and forwards the other two requests to the HRT as special exceptions or interrupts. Because the VMM and HRT may need to share additional information, they share a data page in memory. For a function call request, the page contains a pointer to the function and its arguments at the start and the return code at completion. For an address space merger, the page contains the CR3 of the calling process. The HRT indicates to the VMM when it is finished with the current request via a hypercall.

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

4.4 Merged address spaces

To achieve a merged address space, we leverage the canonical 64-bit address space model of x64 processors, and its wide use within existing kernels, such as Linux. In this

model, the virtual address space is split into a "lower half" and a "higher half" with a gap in between, the size of which is implementation dependent. In a typical process model, e.g., Linux, the lower half is used for user addresses and the higher half is used for the kernel.

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

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

Nautilus additions

In order to support Multiverse in the Nautilus AeroKernel, we needed to make several additions to the codebase. Most of these focus on runtime initialization and correct operation of event channels. When the runtime and application are executing in the HRT, page faults in the ROS portion of the virtual address space must be forwarded. We added a check in the page fault handler to look for ROS virtual addresses and forward them appropriately over an event channel.

One issue with our current method of copying a portion of the PML4 on an address space merger is that we need to keep the PML4 synchronized. We must account for situations in which the ROS changes top-level page table mappings, even though these changes are rare. We currently handle this situation by detecting repeat page faults. Nautilus keeps a per-core variable keeping track of recent page faults, and matches duplicates. If a duplicate is found, Nautilus will re-merge the PML4 automatically. More clever schemes to detect this condition are possible, but unnecessary since it does not lie on the critical path.

For correct operation, Multiverse requires that we catch *all* page faults and forward them to the ROS. That is, if we collect a trace of page faults in the application running native and under Multiverse, the traces should look identical. However, because the HRT runs in kernel mode, some paging semantics (specifically with copy-on-write) change. In default operation, an x86 CPU will only raise a page fault when writing a read-only page in user-mode. Writes to pages with the read-only bit while running in ring 0 are allowed to proceed. This issue manifests itself in the form of mysterious memory corruption, e.g. by writing to the zero page. Luckily, there is a bit to enforce write faults in ring 0 in the `cr0` control register.

Component	*SLOC*			
	C	ASM	Perl	Total
Multiverse runtime	2232	65	0	2297
Multiverse toolchain	0	0	130	130
Nautilus additions	1670	0	0	1670
HVM additions	600	38	0	638
Total	**4502**	**103**	**130**	**4735**

Figure 8: Source Lines of Code for Multiverse.

Before we built Multiverse, Nautilus lacked support for system calls, as the HRT operates entirely in kernel mode. However, a legacy application will leverage a wide range of system calls. To support them, we added a small system call stub handler in Nautilus that immediately forwards the system call to the ROS over an event channel. There is an added subtlety with system calls in HRT mode, as they are now initiating a trap from ring 0 to ring 0. This conflicts with the hardware API for the `SYSCALL`/`SYSRET` pair of instructions. We found it interesting that `SYSCALL` has no problem making this idempotent ring transition, but `SYSRET` will not allow it. The return to ring 3 is unconditional for `SYSRET`. To work around this issue, we must emulate `SYSRET` and execute a direct `jmp` to the saved `rip` stashed during the `SYSCALL`.

While we can build a particular runtime system with the Multiverse toolchain using custom compilation options, this is not possible for the legacy libraries they rely on. We are then forced into supporting the compilation model that the libraries were initially compiled with. While arbitrary compilation does not typically present issues for user-space programs, complications arise when executing in kernel mode. One such complication is AMD's *red zone*, which newer versions of GCC use liberally. The red zone sits *below* the current stack pointer on entry to leaf functions, allowing them to elide the standard function prologue for stack variable allocation. The red zone causes trouble when interrupts and exceptions operate on the same stack, as the push of the interrupt stack frame by the hardware can destroy the contents of the red zone. To avoid this, kernels are typically compiled to disable the red zone. However, since we are executing code in the ROS address space with predetermined compilation, we must use other methods.

In Nautilus, we address the red zone by ensuring that interrupts and exceptions operate on a well known interrupt stack, not on the user stack. We do this by leveraging the x86 Interrupt Stack Table (IST) mechanism, which allows the kernel to assign specific stacks to particular exceptions and interrupts by writing a field in the interrupt descriptor table. `SYSCALL` cannot initiate a hardware stack switch in the same way, so on entry to the Nautilus system call stub, we pull down the stack pointer to avoid destroying any red zone contents.

4.5 Complexity

Multiverse development took roughly 5 person months of effort. Figure 8 shows the amount of code needed to support Multiverse. The entire system is compact and compartmentalized so that users can experiment with other AeroKernels or runtime systems with relative ease. While the codebase is small, much of the time went into careful design of the execution model and working out idiosyncrasies in the hybridization, specifically those dealing with operation in ker-

nel mode.

5. EVALUATION

In this section we evaluate Multiverse using microbenchmarks and 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. 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.

Microbenchmarks

We first evaluate the latency of system call execution in a VM and in a VM with Multiverse. We tested 9 widely used system calls, some of which leverage the Linux `vdso` mechanism. For the `fwrite()`, `read()`, and `mmap()` calls, the functions operate on 1MB of data. Figure 9 shows the results. We can see that the two `vdso` system calls, `getpid()` and `gettimeofday()` both perform slightly better in Multiverse, and we suspect that this slight improvement may come from a sparsely populated TLB on the HRT core.

Overall, forwarding the system call events through the HVM event channel introduces overheads, but these overheads are less important than the utilization of a particular system call, as the ultimate goal is to end up with an HRT in the Native mode of operation. To save development effort, rarely used system calls that are not performance critical can continue to leverage the legacy OS version of the function.

Racket

Racket [16, 15] 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.[1] Recently, support has been added to Racket for parallelism via futures [35] and places [36].

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. These features include complex package management via the filesystem, shared library-based support for native code, JIT compilation, tail-call elimination, live variable analysis (using memory protection), and garbage collection.

Our port of Racket to the HRT model takes the form of an instance of the Racket engine embedded into a simple C program. Racket already provides support for embedding an instance of Racket into C, so it was straightforward to produce a Racket port under the Multiverse framework. This port uses a conservative garbage collector, the SenoraGC, which is more portable and less performant than the default, precise garbage collector. The port was compiled with GCC 4.6.3. The C program launches a pthread that in

[1] http://racket-lang.org

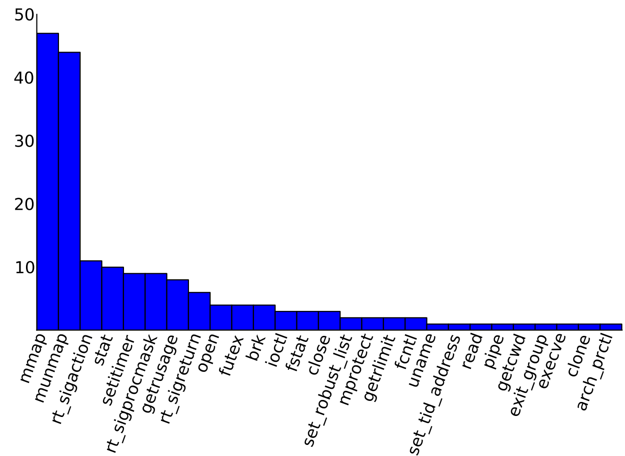

Figure 11: Utilization of system calls in the Racket runtime without any benchmark.

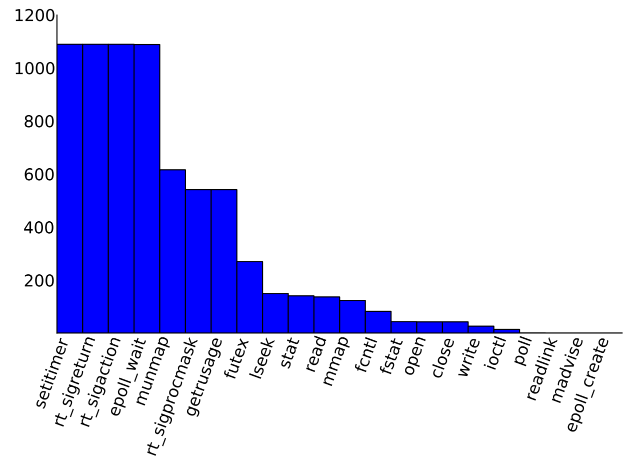

Figure 12: Utilization of system calls in the Racket runtime for a run of the binary-tree-2 benchmark.

turn starts the engine. Combined with the incremental usage model of Multiverse, the result is that the Racket engine executes in the HRT.

When compiled and linked for regular Linux, our port provides either a REPL interactive interface through which the user can type Scheme, or a command-line batch interface through which the user can execute a Scheme file (which can include other files). When compiled and linked for HRT use, our port behaves identically.

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. Figure 10 characterizes these benchmarks from the low-level perspective. 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

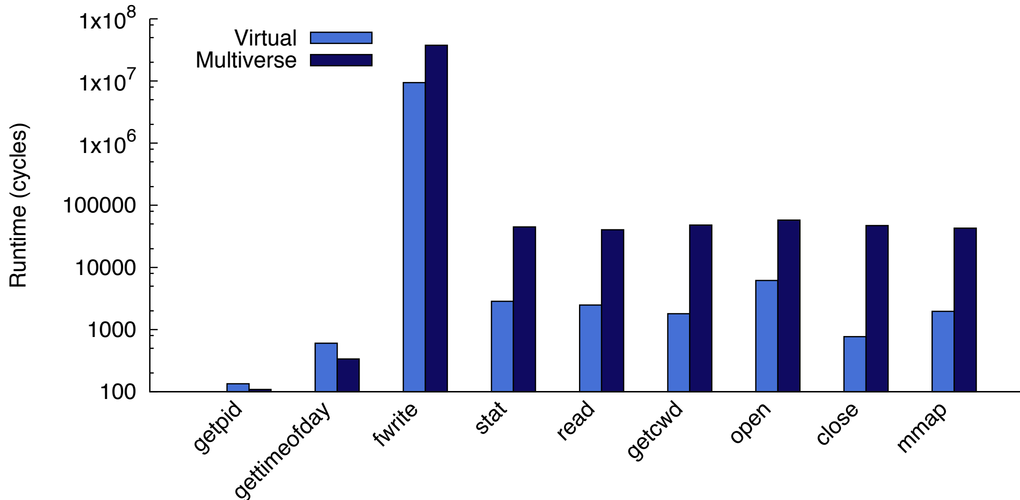

Figure 9: Latency in cycles for system calls running virtual and in Multiverse. These numbers represent round-trip latencies for forwarding system calls from the HRT to the ROS and back. Frequent use of a particular call will suggest a target for transitioning that call to a custom implementation within the AeroKernel.

Benchmark	System Calls	Time (User/Sys) (s)	Max Resident Set (Kb)	Page Faults	Context Switches
fannkuch-redux	1279	2.73/0.01	21284	5358	33
binary-tree-2	1260	31.98/0.10	82072	31082	491
fasta	29989	12.23/0.10	43568	14956	627
fasta-3	35115	31.28/0.17	80492	25418	1075
n-body	18763	41.15/0.19	152300	45064	1430
spectral-norm	23800	39.39/0.24	182300	51452	1695
mandelbrot-2	3667	7.76/0.05	43600	14250	291

Figure 10: System utilization for Racket benchmarks. A high-level language has many low-level interactions with the OS.

fault forwarding mechanisms. We ran all tests using the hardware setup described at the beginning of this section.

Figure 12 describes the number and type of system calls the Racket runtime invoked while running binary-tree-2, a benchmark which creates and traverses trees. This benchmark makes extensive use of garbage collection. We can see the majority of calls are those made in service of the Racket runtime's garbage collection. For example, the `rt_sigaction` and `rt_sigreturn` system events involve setting up for receiving `SIGSEGV` signals due to page faults that drive the garbage collector. The timer, `getrusage() calls`, and polling activity is used to support Scheme-level cooperative threads in the run-time. `mmap()`, `munmap()`, and `mprotect()`, arrange memory protections to create `SIGSEGV`s for the garbage collector. Figure 11 gives a similar breakdown of system calls incurred when setting up the garbage collection environment, as calls to `mmap()` and `munmap()` dominate the system calls for the creation of the heap. This is also seen in Figure 12, as small sections of the heap are frequently freed with calls to `munmap()`.

Figure 13 compares the performance of the Racket benchmarks run natively on our hardware, under virtualization, and as an HRT that was created with Multiverse. The overhead of the Multiverse case compared to the virtualized and native cases is due to the frequent interactions, such as those described above, with the Linux ABI. From the small system times in Figure 10, we can surmise that a large portion of the interactions in these cases likely arise from page faults rather than system calls. In the Multiverse case, these are forwarded from the HRT to the ROS to be handled instead of being handled locally.

One should understand that these results constitute an initial baseline of performance. 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 and behaves identically. To be clear, *all of the Racket runtime except Linux kernel ABI interactions is seamlessly running as a kernel.* This represents a *starting point* for HRT development in the incremental model. The next steps would be to port bottleneck functionality, for example the `mmap()`, `mprotect()`, and signal mechanisms the garbage collector depends on, to kernel mode via AeroKernel, perhaps using AeroKernel overrides. In effect, these comprise page table edits combined with page faults, all of which can occur hundreds of times faster within the kernel instead of behind a system call interface.

6. RELATED WORK

Work on specialized kernels goes back decades, and the design of Nautilus is heavily influenced by much of this early work, including Exokernels [12, 13], SPIN [8], Scout [28], KeyKOS [10], and ADEOS [38].

In a similar vein to Nautilus, Arrakis [31] allows applications full access to hardware through library operating systems linked into their address space. However, Arrakis aims to reduce OS overhead related to I/O, and does not al-

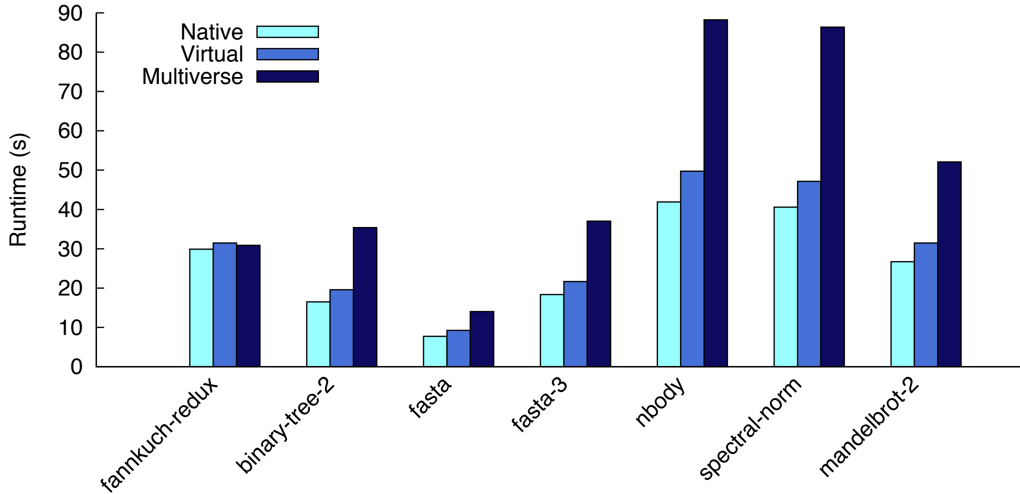

Figure 13: Performance of Racket benchmarks running Native, Virtual, and in Multiverse. Note that the Multiverse result is the result of Multiverse's automatic hybridization of Racket—it is the *starting point* for incremental enhancement within the HRT model.

low users to boot an application or runtime into an entirely specialized OS environment.

The nonkernel [7] similarly enables unprivileged user applications to access hardware directly using virtualization support, but targets fine-grained resource provisioning rather than runtime system performance and support. Unikernels [27] and OSv [24] also rely on thin hypervisor layers to achieve low latency and predictability for cloud applications.

Drawbridge [32] introduces a lightweight *picoprocess* model wherein sandboxed applications run within a Windows libOS. The picoprocess interacts with a virtualization layer through a thin and highly abstracted ABI that simplifies the implementation of OS code in the picoprocess.

While the above systems explore aspects of specialized operating systems, none of them provide a mechanism by which an application can leverage both existing OS functionality and functionality from a specialized kernel.

The Dune system [6] allows a special kernel module to promote selected processes to ones that can access privileged CPU features on a legacy Linux system. Dune leverages virtualization support to give applications the ability to access, for example, page tables and protection hardware. While Dune gives applications access to previously unavailable hardware, it does so from within the context of a Linux process. Unlike Multiverse, Dune does not give the application the capability to run in an entirely separate OS.

Libra [3] bears similarities to our system in its overall architecture. A Java Virtual Machine (JVM) runs on top of the Libra libOS, which in turn executes under virtualization. A general-purpose OS runs in a *controller partition* and accepts requests for legacy functionality from the JVM/Libra partition. This system involved a manual port, much like our previous paper. However, the HVM gives us a more powerful mechanism for sharing between the ROS and HRT as they share a large portion of the address space. This allows us to leverage complex functionality in the ROS like shared libraries and symbol resolution. Furthermore, the Libra system does not provide a way to automatically create these specialized JVMs from their legacy counterparts.

The Blue Gene/L series of supercomputer nodes run with a Lightweight Kernel (LWK) called the Blue Gene/L Run Time Supervisor (BLRTS) [2] that shares an address space with applications and forwards system calls to a specialized I/O node. While the bridging mechanism between the nodes is similar, there is no mechanism for porting a legacy application to BLRTS. Others in the HPC community have proposed similar solutions that bridge a *full-weight* kernel with an LWK in a hybrid model. Examples of this approach include mOS [37], ARGO [5], and IHK/McKernel [34]. The Pisces Co-Kernel [30] treats performance isolation as its primary goal and can partition physical hardware between *enclaves*, or isolated OS/Rs that can involve different specialized OS kernels.

In contrast to the above systems, our HRT model is the only one that allows a runtime to act *as* a kernel, enjoying full privileged access to the underlying hardware. Furthermore, as far as we are aware, none of these systems provide an automated mechanism for producing an initial port to the specialized OS/R environment.

7. 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 like it runs just 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.

We plan to extend Multiverse to work with a wider range of real-world runtime systems, especially parallel runtime systems like Legion. We also intend to explore the transition between our Incremental model and the Accelerator model, using Multiverse to identify bottlenecks and replace legacy functionality with optimized versions in the Nautilus AeroKernel. Finally, we hope to investigate radically different execution groups that include execution contexts other than threads.

8. REFERENCES

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

[2] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. Castaños, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss. An overview of the blue gene/l system software organization. In *Proceedings of the Euro-Par Conference on Parallel and Distributed Computing (EuroPar 2003)*, Aug. 2003.

[3] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE 2007)*, pages 44–54, June 2007.

[4] 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.

[5] P. Beckman. Argo: An exascale operating system. http://www.mcs.anl.gov/project/argo-exascale-operating-system.

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

[7] M. Ben-Yehuda, O. Peleg, O. Agmon Ben-Yehuda, I. Smolyar, and D. Tsafrir. The nonkernel: A kernel designed for the cloud. In *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSYS 2013)*, July 2013.

[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 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 267–283, Dec. 1995.

[9] 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.

[10] 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.

[11] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013-4744, Sandia National Laboratories, June 2013.

[12] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th 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 15th ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 251–266, Dec. 1995.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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. In Submission.

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

[20] 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.

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

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

[23] 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.

[24] 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.

[25] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the* 24th *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Apr. 2010.

[26] 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* 1st *USENIX Conference on Hot Topics in Parallelism (HotPar 2009)*, pages 10:1–10:6, Mar. 2009.

[27] 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* 18th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 461–472, Mar. 2013.

[28] 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* 5th *Workshop on Hot Topics in Operating Systems (HotOS 1995)*, pages 58–61, May 1995.

[29] J. Oayang, B. Kocoloski, J. Lange, and K. Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the 24th International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)*, June 2015.

[30] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the* 24th *International Symposium on High-Performance Parallel and Distributed Computing*, pages 149–160, June 2015.

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

[32] 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* 16th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 291–304, Mar. 2011.

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

[34] T. Shimosawa, B. Gerofi, M. Takagi, G. Nakamura, T. Shirasawa, Y. Saeki, M. Shimizu, A. Hori, and Y. Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC 2014)*, Dec. 2014.

[35] 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.

[36] 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.

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

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