Towards a Practical Ecosystem of Specialized OS Kernels

Conghao Liu and Kyle C. Hale
Illinois Institute of Technology
{cliu115@hawk,khale@cs}.iit.edu

ABSTRACT

Specialized operating systems have enjoyed a recent revival driven both by a pressing need to rethink the system software stack in several domains and by the convenience and flexibility that on-demand infrastructure and virtual execution environments offer. Several barriers exist which curtail the widespread adoption of such highly specialized systems, but perhaps the most consequential of them is that these systems are simply *difficult to use*. In this paper we discuss the challenges faced by specialized OSes, both for HPC and more broadly, and argue that what is needed to make them practically useful is a reasonable development and deployment model that will form the foundation for a *kernel ecosystem* that allows intrepid developers to discover, experiment with, contribute to, and write programs for available kernel frameworks while safely ignoring complexities such as provisioning, deployment, cross-compilation, and interface compatibility. We argue that such an ecosystem would allow more developers of highly tuned applications to reap the performance benefits of specialized kernels.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Development frameworks and environments**; *Virtual machines*; *Software architectures*;

ACM Reference Format:
Conghao Liu and Kyle C. Hale. 2019. Towards a Practical Ecosystem of Specialized OS Kernels. In *9th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS'19), June 25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3322789.3328742

1 INTRODUCTION

Several recent trends have prompted a reopening of the discussion on limitations of general-purpose OSes. Increasing hardware heterogeneity [45, 48] poses significant challenges for system software aiming to support a wide array of applications efficiently [32]. An increasing diversity of applications means that a general-purpose OS must be all things to all users, potentially sacrificing well-matched abstractions and mechanisms. Predictable operation and performance, while sometimes detrimental to resilience against exploitation [43], can be very important for certain applications [36], but is often elusive for general-purpose systems [10, 18, 34]. Unmet needs for fine-grained task abstractions [37], e.g. to enable serverless computing [27], have sparked new specialized system designs such as ZygOS [41] and Amazon's Firecracker VMM.[1]

Specialized OSes (SOSes) provide one avenue for addressing these challenges, where here we broadly construe a specialized OS as one tailored for a specific workload or class of workloads. This includes library OSes, Unikernels, and light-weight kernels. While they have been discussed in the literature for many years [2, 6, 8, 33], their recent resurgence is in part due to the availability and ease-of-use of commodity virtualization software and public cloud platforms like Amazon's AWS and Chameleon [20]. One of the early visions for virtualization technology—namely, practical OS experimentation—is now coming true. There is now a wide array of specialized OSes available today, from kernels designed for the cloud such as OSv [23], Arrakis [39], and Mirage [31] to OS designs to support language and hybrid parallel runtimes [1, 15, 16, 19]. Many efforts focus on extreme scalability at both the intra-node and inter-node level, including Barrelfish [3], Andromeda [44], and Corey [7]. Lightweight kernels (LWKs) specifically designed for raw performance have been around in HPC for more than a decade [13, 21, 28], and the HPC community is now also looking at Unikernels [29], in addition to multi-kernel and co-kernel approaches [4, 12, 38, 47]. Benefits of specialized kernels include their small size, their performance, predictability, and in some cases security. Unikernels and LWKs can make virtualization more attractive, as their execution environment can be more hypervisor-friendly [24].

Despite their benefits, SOSes still face several challenges. The designers must make the decision whether or not the kernel interface will retain POSIX compatibility (or binary compatibility with, e.g. Linux), pick the right abstractions for the target workload(s), and decide on the right level of protection, among other issues. Specialization for its own sake is not necessarily a good idea, and as work in the architecture community shows, striking a good balance between domain-specific design and general-purpose abstractions can pay off [35]. Some of these design points can (and should) be based on foundational principles, but others require experimentation and design iteration.

However, because SOSes often eschew the usual interfaces, and because their build toolchains and supported hardware vary, they tend to be *difficult to use*[2]. This difficulty, of course, impedes the

This project is made possible by support from the United States National Science Foundation (NSF) via grants CNS-1718252 and CNS-1763612.

ROSS'19, June 25, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6755-4/19/06…$15.00
https://doi.org/10.1145/3322789.3328742

[1] https://firecracker-microvm.github.io/
[2] One could, of course, make the argument that this is true for any OS, but commodity OSes with momentum in the community (with industry and open-source support) have already by necessity built up an ecosystem of support tools. Take, for example,

progress of OS experimentation and can discourage developers from getting involved in kernel development.

While others have worked toward making the kernels themselves easier to build [42], we contrast the current state of affairs for writing a program for, e.g. a Unikernel, with starting a new `cargo` project in Rust (with several simple commands, you can easily initialize/build/run your project. Package dependencies, compilation dependencies will be automatically handled by the cargo system)—starting from the outset with ecosystem integration in mind.

We propose a new development model for SOSes which is rooted in the idea of building an *ecosystem* for OS kernels. Much like the public ecosystems built around container images and virtual machine images have been a boon to those technologies, we believe a sound development model paired with a vibrant kernel ecosystem will not only encourage developers to more actively experiment with specialized OSes and Unikernels, but will also make them a more practically useful tool.

A key research challenge for tools which support such an ecosystem, however—especially for HPC—is the preservation of performance gains given by the SOS.

2 MOTIVATION

While it is easier than it has ever been to build, debug, experiment with, and deploy specialized OSes (e.g. using IaaS clouds, QEMU, IPMI, kgdb, etc.), writing programs for them is often very cumbersome, even when the kernel maintains binary compatibility with a more popular OS. We are currently involved in development of an SOS kernel called Nautilus[3] which is designed to be paired with high-performance, parallel-runtime systems [15, 16]. When writing (or porting) programs or runtime systems for Nautilus, the developer currently has to add invocation hooks into the kernel's initialization code and manually integrate their code into the kernel build system. This is obviously cumbersome, as it requires developers to modify a kernel codebase and in some cases deal with the intricacies of its build system.

Building and running a program for OSv is simpler. OSv has a convenient tool called Capstan which makes writing and deloying new applications easier.

```
$> capstan package init \
    --name "java-example"
```

This command initializes a new Capstan app package in the current directory. Then the user can start writing code for the new program. Once finished, the user can run:

```
$> capstan package compose \
    -p java-example
```

This command fuses the application and kernel into a bootable OSv QCOW2 image. To run it, we do the following:

```
$> capstan run java-example \
    -p qemu --boot default
```

In addition to the convenience of developing and deloying new apps, OSv also provides scripts to help users tailor, generate, or publish new kernel images to Capstan, Google Cloud Storage, or Amazon AWS. However, there is no way to integrate *other* specialized OSes, or to use complicated deployment modes as discussed in Section 4. Similar inspiration also comes from Rust. Consider building a program with the Rust[4] programming language. The Rust developers have developed a build tool called `cargo` which allows programmers to write code in a way that is amenable to testing and release. For example, rather than writing a program and then manually preparing the code for release (e.g. by providing a `configure` script and integrating with Autotools and the GNU Build System), the Cargo system sets the programmer up from the start for releasing their code and publishing it to an ecosystem of Rust packages[5]. The below shows the series of invocations needed to create, test, and publish a project using this system:

```
$> cargo new my-program
$> cd my-program
...development...
$> cargo build
$> cargo test
$> cargo package
$> cargo publish
```

This program could then be run (and performance measured) in a specialized software environment, for example using containers. Our vision is to combine these approaches to promote an ecosystem of specialized OS kernels, where steps above would correspond to writing an application for a specific OS or a component of that OS, and subsequently deploying that OS on virtual or physical hardware. We claim such an ecosystem (and systems to support it) should ideally meet the following requirements.

Discoverability: It should be easy for developers to find kernels which fit their particular needs, for example systems designed for application sandboxing (e.g. Drawbridge [40]), kernels for the cloud (OSv [23]), or kernels for HPC (HermitCore [29], Kitten [28], IHK/McKernel [12], Nautilus [15], mOS [47]). Ideally images would be tagged and searchable. This would look very similar to existing ecosystems for VM disk images (VMware's virtual appliance marketplace) or container images (Docker Hub).

Ease-of-Use: When using a kernel image, it should not be necessary for the developer to understand kernel internals. Complexities of the build toolchain and deployment should also be abstracted away by default when possible. Advanced users, however, and developers wishing to augment a kernel should be given the option.

Composability: Users should be able to build pipelined workflows using *different kernels* deployed in *different ways*. This allows users to build complex functionality out of basic building blocks, where here the building blocks are app/kernel invocations. This presents a challenge because (1) it requires a *standard* communication substrate and messaging protocol between kernels and (2) the deployment tool must be able to reconcile workflow structure with the specific kernel invocations while maximizing parallelism. For example, one kernel invocation might involve a multi-kernel approach which uses most physical cores on the machine, while another might only use a unikernel on a single physical core. The deployment tool then must play the role of job scheduler. For (1), once the communication mechanisms are decided uopn (e.g. IPIs

the slow evolution of the Multiboot2 standard (very useful for SOSes, but Linux does not use it).

[3] https://github.com/hexsa-lab/nautilus

[4] https://www.rust-lang.org/

[5] https://crates.io/

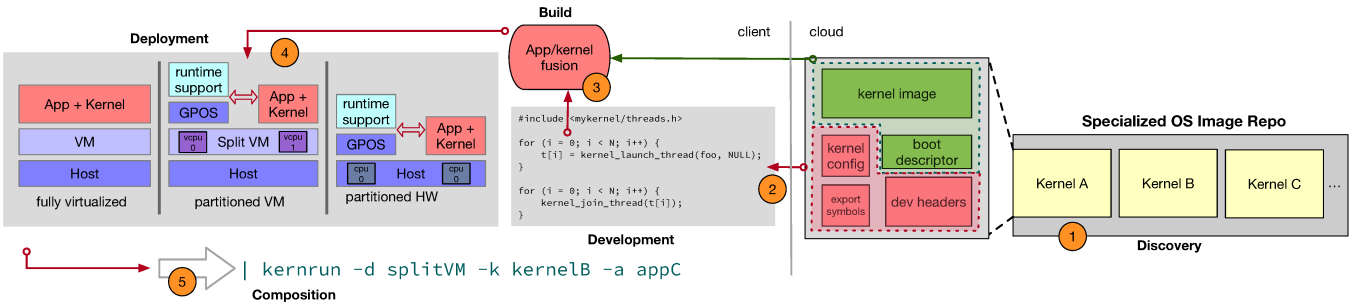

Figure 1: Overview of our proposed model.

with a page of shared memory), we can force compilation of the kernel with a static library. However, OS developers still must provide hooks into this communication library. For (2), we can leverage prior art on job scheduling in heterogeneous systems and take inspiration from workflow languages, e.g. Swift [46].

Customizability: Advanced developers should have a way to rapidly modify existing parts of a kernel, contribute their own components, test them, and deploy them (much like `cargo`, as above). We contrast this with the laborious process for building, testing, and deploying a Linux kernel, which is mostly manual.

Performance: Systems involved (e.g. build tools and deployment support runtimes) should have very little performance overhead. This is critically important for HPC applications running in a large-scale environment.

Figure 1 depicts an example of an app/kernel (e.g. Unikernel) development workflow. Kernel images are stored in a publicly searchable catalog (1). A developer then pulls one of these kernels down (2) to begin working on a project. While this might just be one invocation of a build tool, everything necessary to do development for this particular kernel, such as kernel headers, exported symbols, and other configuration and metadata is pulled down transparently for the developer. The developer might specify a particular kernel version or kernel configuration in this invocation (e.g. `pull nautilus:infiniband`). Once the developer has finished writing the app or kernel component, the build toolchain is invoked to perform any necessary app + kernel compilation and linking (3). This might only be necessary if using a kernel that does not include support for separately compiled code and dynamic linking. Once this is complete, the user can deploy the combination in a variety of ways (4). We discuss these deployment modes in more detail in Section 4. The output of one kernel invocation can be composed with another to enable pipeline-oriented workflows (5).

In addition to being able to pull *existing* kernel versions or configurations, developers should be able to create their own (e.g. forks of existing images) either by changing configurations or modifying the kernel code directly. Once created they can be published in the public catalog using the build tool (again, much like cargo publish). Mechanisms for easing the pain of debugging kernel code (e.g. automatic linking of gdb stubs, IPMI/Redfish integration, and serial console support) should be available as well.

3 DIVER

Driven by the requirements in Section 2, we developed Diver, a development, compilation, and deployment toolchain which aims to help users discover kernel images, develop and deploy new applications, and tailor and publish new kernel images.

Diver is a client-side tool which works in a similar fashion to Cargo and Capstan, and which implements a specialized kernel workflow as shown in Figure 1. It can be used to search for kernels by tags based on users' needs from a catalog server. Once the kernel name is known, it can can be used to fetch the target kernel image. Diver can also initiate an app/runtime development environment based on the particular kernel. Diver automatically generates a Makefile template for the new app, and downloads all necessary kernel files like headers, symbol tables, and bootloader configurations. When users are finished with development, Diver can help build and boot the app using different deployment modes.

Diver also supports uploading new kernel images to a catalog server. Kernel publishers must follow Diver's requirements, providing necessary scripts for generating kernel images, configuring kernels, testing, and deploying applications. One of our primary goals with Diver is to develop standard interfaces for development/management tools aimed at specialized OSes. We take build and deployment tools like Capstan and Cargo as inspiration and strive to ease unnecessary burdens from developers and users of these systems.

Figure 2 depicts a workflow using Diver. The user first creates a development environment. They can then create an app/kernel combination (which we call a "net") using `diver build`. Finally, the user can deploy the net using a chosen mode (in this case a partitioned VM) with the "dive" command. While Diver currently supports just Nautilus and OSv, we plan to extend it to other specialized OSes. Our Diver prototype and its code will be freely available when this paper is published.

4 INTEGRATING DEPLOYMENT MODES

Once a specialized kernel is built and made ready to boot (paired with an application or runtime system), the user then must choose how to run (deploy) it. Existing tools simply launch the SOS in a VM, but for HPC systems, which might involve complex multi-kernel environments, a simple VM-based deployment may not be sufficient. We thus must support different ways of using the underlying hardware. When deploying with Diver, our goal is to hide

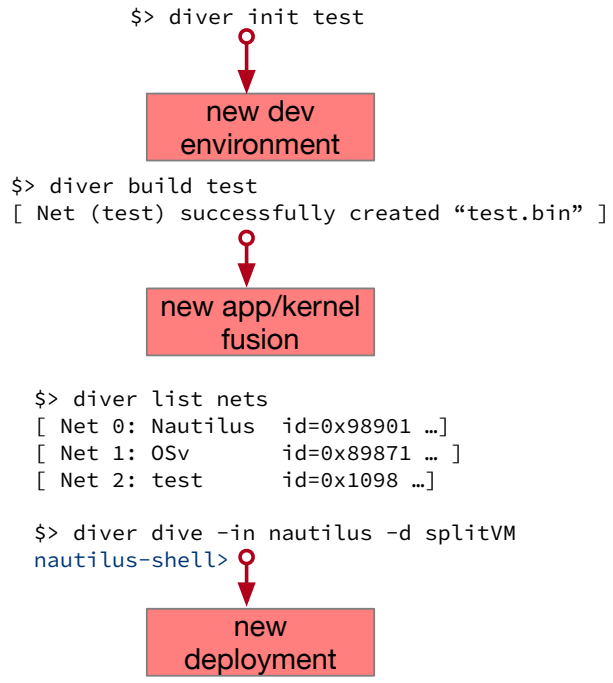

Figure 2: An example workflow using Diver.

as of this complexity as possible *by default*, but allow advanced users to customize the deployment. For example, we put in place a sane default deployment mode (e.g., fully virtualized), but the user can choose a different mode for each app/kernel invocation. Users will also be able to customize options for machine configuration. For example, in a virtualized deployment mode, users can choose attached devices, passthrough configurations, virtual disks, console options, etc. This will be very similar to libvirt invocations (e.g. with `virsh`). With native deployment modes, users will be able to specify resource partitioning (e.g. physical core distribution, physical memory map, shared address space layout, etc.).

Figure 1(4) shows three possible modes of deployment. We now describe these modes and outline how Diver integrates with them.

4.1 Fully Virtualized

This deployment mode (left side of Figure 1(4)) puts the specialized OS and app combination in its own virtual machine. This is the most commmon model that many specialized kernels support, especially Unikernels, which are designed with paravirtualization (i.e. only virtio device drivers) in mind. Our current Diver prototype supports this model. Invocations are similar to libvirt tools, and the backend hypervisor can be configured, e.g. qemu-kvm [5, 22] or Palacios [28]. To support this deployment mode, the specialized OS need only support automatic shutdown (i.e., a kernel bootup, application invocation, kernel shutdown sequence rather than an always-on mode of operation). This will ensure that several kernel invocations can be composed properly. Ideally the kernel would also support debugging stubs for integration with Diver.

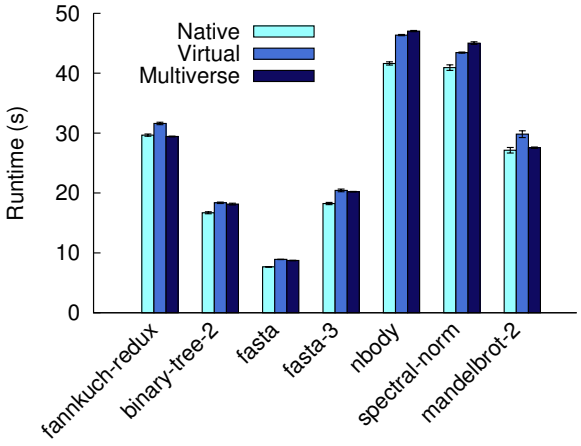

Figure 3: Language shootout benchmark performance with Racket runtime running native, in a virtual machine, and a VM split between two OSes (using Multiverse).

4.2 Partitioned VMs

In this mode (middle portion of Figure 1(4)), a virtual machine is space-partitioned between two operating systems. One is a general-purpose OS (GPOS in the figure) such as Linux. This OS serves the role of fielding forwarded requests for functionality not supported by the specialized OS. For example, a Unikernel with no filesystem support might forward syscalls to the GPOS to be serviced. This also gives the specialized kernel a way to use devices while relying on the device drivers of the GPOS (much like a Dom0 VM in Xen). Libra [1] first used this mode for a JVM-specific kernel. Unlike a Dom0 setup, however, the virtual cores and memory of the VM are *space-shared* between the OSes. This gives an opportunity for more efficient communication and interesting superpositions of OS state. We previously explored this mode using Hybrid Virtual Machines (HVM), which allowed us to share portions of the virtual address space between kernels and run a user program in a *split execution environment* between the kernels [16]. Namely, the "high-half" (kernel) of the address space are distinct, and the "low-half" (user) of the address space is shared. A runtime system called Multiverse [17] paired with an HVM allows legacy parallel programs (for Linux) to be automatically transformed to work with this model.

This type of deployment mode requires paravirtual support (hypercalls) for communication between kernels, and if state superpositions are to be supported, special handling for them.

As we pointed out in Section 2, underlying deployment modes should not introduce significant overheads for systems booted with Diver. Figure 3 shows a performance comparison for the The Language Benchmarks Game for the Racket language on an 8-core AMD system. Here we compare the performance of the benchmarks running on Linux, running on a Linux VM, and a running on a version of the Racket runtime system that has automatically been ported to the partitioned VM mode using Multiverse. There is little overhead (on the order of a few thousand cycles), and this

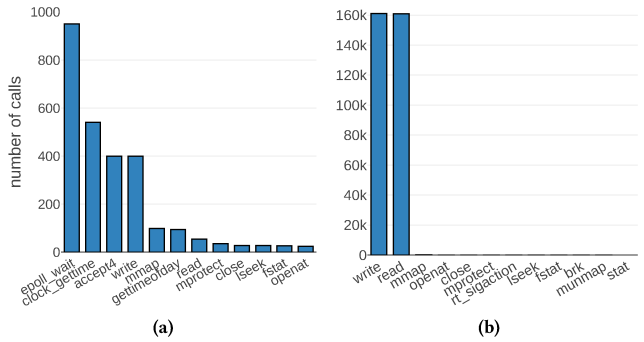

Figure 4: Histograms representing syscall invocation trace for memcached and bzip2.

arises due to forwarding between kernels. In general, an application or runtime's heavy reliance on the legacy (e.g. Linux) ABI will increase the number of these forwarded events, and will thus affect performance. As an example, Figure 4 shows a breakdown of a system call trace for memcached (a) and bzip2 (b). Memcached does has a more varied distribution of syscalls, but fewer instances of these invocations. This indicates that forwarding overhead would be minimal for this deployment mode. Bzip2, however, has several hundred thousand invocations of `read()` and `write()`. This benchmark invocation runs for about two minutes, which means that the overall performance may still be acceptable depending on forwarding overheads (usually on the order of 1000 μsec for efficient shared memory communication). A more complete analysis of forwarding events for this type of deployment (specifically for scientific workloads) can be found in [14]. While the performance of the underlying deployment mode is somewhat orthogonal to Diver itself, we must ensure that Diver does not introduce any *further* overheads, e.g. by scheduling kernel invocations on top of one another, or by introducing interference by space-sharing the machine inappropriately.

Based on the observation from our syscall breakdown experiments, here we argue that one of the important use cases of this partitioned VM model is incremental porting of legacy programs. Porting existing applications from Linux to another OS that is not ABI compatible with Linux requires huge engineering effort. With the help of this partitioned VM model, developers are able to selectively rewrite the code related to those syscalls of the biggest concern in terms of performance (e.g. epoll_wait in Figure 4 (a)).

4.3 Partitioned Hardware

This mode (right side of Figure 1(4)) is similar to the one discussed in the previous section, but allows the cores, memory, and devices of the physical hardware to be partitioned between a GPOS and a specialized kernel. Lange et al. explored this model using the Pisces Co-kernel architecture [38] and the XEMEM system for efficiently sharing memory between kernels [25]. These systems help to support efficient, in-node application composition [9, 26]. The main requirement with this mode is that the GPOS must support offlining cores, and the specialized OS must support bootup in a special software environment. Both kernels in this mode must have some mechanism for inter-kernel communication and memory sharing. Hardware for core isolation (as in on the Blue Gene/L [11]) makes things easier.

5 CHALLENGES AND FUTURE WORK

While we have an existing prototype, realizing the ambitious vision laid out in Section 2 will involve addressing several challenges. First, we must integrate the more complex (partitioned) deployment modes into Diver. We must also add support for more specialized kernels, such as IHK/McKernel and HermitCore. An ultimate goal is to develop standard features and interfaces that kernels must implement to fit in with the Diver system. Supporting efficient composition of app/kernel invocations (e.g., piping the output of one or several app/kernel invocation to another as shown in Figure 1(5)), where both the app/kernel combination and the deployment mode may vary, also presents a challenge. We also plan to extend the deployment model to include support for heterogeneous systems with various accelerators and devices.

6 RELATED WORK

UniK[6] from solo.io is the only tool we are aware of that is similar to Diver. It serves as a glue between user applications and unikernels. UniK helps to compile application source code into a unikernel (using lightweight bootable disk images) and lightweight virtual machines rather than traditional application binaries. It utilizes a simple Docker-like command line interface, making building unikernels as easy as building containers. UniK runs and manages instances of compiled images across a variety of cloud providers as well as locally using VMs.

Unlike Diver, however, UniK is primarily focused on cloud environments. The following are some major differences. (1) Performance is not the primary concern for UniK. It only supports running the application in a VM (either on cloud or locally). Diver supports three different deployment models to allow the user to make tradeoffs between performance and flexibility. (2) UniK does not consider the composition of multiple kernel/app combinations. (3) UniK focuses on the compilation (image generation) process. It does not help in setting up a sane development environment for the user, which includes searching for and downloading necessary unikernel header files and preparing a default build toolchain.

EbbRT is a framework for building per-application library operating systems [42]. It consists of a set of components called Elastic Building Blocks (Ebbs) that developers can extend, replace, or discard to construct and deploy a particular application. EbbRT provides an event-driven execution environment and a minimal abstraction over the hardware, allowing applications to directly interact with hardware resources. EbbRT splits applications across both specialized and general-purpose OSes. Thus, functionality (such as system calls) can be offloaded. With this replaceable and modularized Elastic Building Blocks design, EbbRT makes libOSes easier to build. However, EbbRT does not target other types of SOSes (e.g. lightweight kernels), and does not consider different split execution environments. Furthermore, they do not consider kernel discovery or kernel composition.

[6] https://github.com/solo-io/unik

Jitsu [30] is a system for securely managing multi-tenant networked applications on embedded infrastructure. It utilizes fast shared-memory channels to provide services that launch MirageOS unikernels in VMs in response to network traffic. Jitsu is designed for Xen/ARM and it modifies the Xen toolstack to lower resource overheads of manipulating virtual machines. Jitsu is similar to Diver in that they both help to run and manage unikernels in VMs, but they do not consider disparate kernels or complex deployment modes.

7 CONCLUSIONS

Both pressing needs for rethinking the software stack and the widespread availability of virtual, on-demand infrastructure have led to a resurgence of specialized operating systems. We argue that now is the time to begin building *ecosystems* for these SOSes to encourage experimentation and design iteration. We discussed requirements that we believe tools to support this ecosystem should meet, and we presented a prototype of such a tool called *Diver* which we hope will be a step towards a specialized OS ecosystem. While Diver currently supports deploying SOSes on virtual infrastructure, we plan to extend the toolchain to support more non-traditional deployment modes including physically partitioned hardware and partitioned virtual machine environments. We further plan to explore coordination between kernel invocations which utilize different deployment modes.

REFERENCES

[1] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a JVM in a virtualized execution environment. In *Proceedings of the* 3^{rd} *International Conference on Virtual Execution Environments*, VEE '07, pages 44–54, June 2007.

[2] T. E. Anderson. The case for application-specific operating systems. In *Proceedings of the* 3^{rd} *Workshop on Workstation Operating Systems*, Apr. 1992.

[3] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the* 22^{nd} *ACM Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, Oct. 2009.

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

[5] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of 2005 USENIX Annual Technical Conference*, USENIX ATC'05, pages 41–46, Apr. 2005.

[6] 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 '95, pages 267–283, Dec. 1995.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the* 8^{th} *USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Dec. 2008.

[8] 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 '95, pages 251–266, Dec. 1995.

[9] N. Evans, K. Pedretti, B. Kocoloski, J. Lange, M. Lang, and P. G. Bridges. A cross-enclave composition mechanism for exascale system software. In *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, June 2016.

[10] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of Supercomputing*, SC '08, Nov. 2008.

[11] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2):195–212, Mar. 2005.

[12] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. On the scalability, performance isolation and device driver transparency of the IHK/McKernel hybrid lightweight kernel. In *Proceedings of the* 30^{th} *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16, pages 1041–1050, May 2016.

[13] 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 '10, Nov. 2010.

[14] R. Gioiosa, R. W. Wisniewski, R. Murty, and T. Inglett. Analyzing system calls in multi-OS hierarchical environments. In *Proceedings of the* 5^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, June 2015.

[15] 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 '15, June 2015.

[16] K. C. Hale and P. A. Dinda. Enabling hybrid parallel runtimes through kernel and virtualization support. In *Proceedings of the* 12^{th} *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'16, pages 161–175, Apr. 2016.

[17] K. C. Hale, C. Hetland, and P. A. Dinda. Multiverse: Easy conversion of runtime systems into os kernels via automatic hybridization. In *Proceedings of the* 14^{th} *IEEE International Conference on Autonomic Computing*, ICAC'17, July 2017.

[18] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of Supercomputing*, SC '10, Nov. 2010.

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

[20] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth. Chameleon: a scalable production testbed for computer science research. In J. Vetter, editor, *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 3 of *Chapman & Hall/CRC Computational Science*, chapter 5. CRC Press, 1 edition, 2018.

[21] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting*, CUG'05, May 2005.

[22] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, June 2007.

[23] 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'14, June 2014.

[24] B. Kocoloski and J. Lange. Better than native: Using virtualization to improve compute node performance. In *Proceedings of the* 2^{nd} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '12, June 2012.

[25] B. Kocoloski and J. Lange. XEMEM: Efficient shared memory for composed applications on multi-os/r exascale systems. In *Proceedings of the* 24^{th} *International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 89–100, June 2015.

[26] B. Kocoloski, J. Lange, H. Abbasi, D. E. Bernholdt, T. R. Jones, J. Dayal, N. Evans, M. Lang, J. Lofstead, K. Pedretti, and P. G. Bridges. System-level support for composition of applications. In *Proceedings of the* 5^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, June 2015.

[27] R. Koller and D. Williams. Will serverless end the dominance of linux in the cloud? In *Proceedings of the* 16^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 169–173, May 2017.

[28] 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'10, Apr. 2010.

[29] S. Lankes, S. Pickartz, and J. Breitbart. HermitCore: A unikernel for extreme scale computing. In *Proceedings of the* 6^{th} *International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS'16, June 2016.

[30] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, 2015.

[31] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the* 18^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 461–472, Mar. 2013.

[32] K. S. McKinley. The yin and yang of hardware heterogeneity: Can software survive? In *Proceedings of the Companion Publication for the ACM SIGPLAN Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Keynote)*, SPLASH '13, pages 1–2, Oct. 2013.

[33] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the* 5^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '95, pages 58–61, May 1995.

[34] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *Proceedings of the* 25^{th} *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 852–863, May 2011.

[35] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *Proceedings of the* 22^{nd} *IEEE International Symposium on High Performance Computer Architecture*, HPCA '16, pages 27–39, Mar. 2016.

[36] K. Ousterhout, C. Canel, M. Wolffe, S. Ratnasamy, and S. Shenker. Performance clarity as a first-class design principle. In *Proceedings of the* 16^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 1–6, May 2017.

[37] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Proceedings of the* 14^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '13, May 2013.

[38] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the* 24^{th} *International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 149–160, June 2015.

[39] S. Peter and T. Anderson. Arrakis: A case for the end of the empire. In *Proceedings of the* 14^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '13, May 2013.

[40] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the* 16^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 291–304, Mar. 2011.

[41] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the* 26^{th} *Symposium on Operating Systems Principles*, SOSP '17, pages 325–341, Oct. 2017.

[42] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: A framework for building per-application library operating systems. In *Proceedings of the* 12^{th} *USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 671–688, Oct. 2016.

[43] R. Sun, D. E. Porter, D. Oliveira, and M. Bishop. The case for less predictable operating system behavior. In *Proceedings of the* 15^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '15, May 2015.

[44] N. Vasilakis, B. Karel, and J. M. Smith. From lone dwarfs to giant superclusters: Rethinking operating system abstractions for the cloud. In *Proceedings of the* 15^{th} *Workshop on Hot Topics in Operating Systems*, HotOS '15, May 2015.

[45] A. Venkat and D. M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Proceedings of the* 41^{st} *Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, June 2014.

[46] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *IEEE Computer*, 42(11):50–60, Nov. 2009.

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

[48] M. Zahran. Heterogeneous computing: Here to stay. *ACM Queue*, 14(6), Dec. 2016.