acmqueue

Unikernels: Rise of the Virtual Library Operating System

Anil Madhavapeddy and David J. Scott

What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?

Cloud computing has been pioneering the business of renting computing resources in large data centers to multiple (and possibly competing) tenants. The basic enabling technology for the cloud is operating-system virtualization such as Xen[1] or VMWare, which allows customers to multiplex VMs (virtual machines) on a shared cluster of physical machines. Each VM presents as a self-contained computer, booting a standard operating-system kernel and running unmodified applications just as if it were executing on a physical machine.

A key driver of the growth of cloud computing in the early days was *server consolidation*. Existing applications were often installed on physical hosts that were individually underutilized, and virtualization made it feasible to pack them onto fewer hosts without requiring any modifications or code recompilation. VMs are also managed via software APIs rather than physical actions. They can be centrally backed up and migrated across different physical hosts without interrupting service. Today commercial providers such as Amazon and Rackspace maintain vast data centers that host millions of VMs. These cloud providers relieve their customers of the burden of managing data centers and achieve economies of scale, thereby lowering costs.

While operating-system virtualization is undeniably useful, it adds yet another layer to an already highly layered software stack now including: support for old physical protocols (e.g., disk standards developed in the 1980s, such as IDE); irrelevant optimizations (e.g., disk elevator algorithms on SSD drives); backward-compatible interfaces (e.g., POSIX); user-space processes and threads (in addition to VMs on a hypervisor); and managed-code runtimes (e.g., OCaml, .NET, or Java). All of these layers sit beneath the *application code*. Are we really doomed to adding new layers of indirection and abstraction every few years, leaving future generations of programmers to become virtual archaeologists as they dig through hundreds of layers of software emulation to debug even the simplest applications?[5,18]

THE COMPILER SOLUTION

This problem has received a lot of thought at the University of Cambridge, both at the Computer Laboratory (where the Xen hypervisor originated in 2003) and within the Xen Project (custodian of the hypervisor that now powers the public cloud via companies such as Amazon and Rackspace). The solution—dubbed *MirageOS*—has its ideas rooted in research concepts that have been around for decades but are only now viable to deploy at scale since the availability of cloud-computing resources has become more widely available.

The goal of MirageOS is to restructure entire VMs—including all kernel and user-space code—into more modular components that are flexible, secure, and reusable in the style of a library operating system. What would the benefits be if *all* the software layers in an appliance could be

compiled within the same high-level language framework instead of dynamically assembling them on every boot? First, some background information about appliances, library operating systems, and type-safe programming languages.

THE SHIFT TO SINGLE-PURPOSE APPLIANCES

A typical VM running on the cloud today contains a full operating-system image: a kernel such as Linux or Windows hosting a primary application running in user space (e.g., MySQL or Apache), along with secondary services (e.g., syslog or NTP) running concurrently. The generic software within each VM is initialized every time the VM is booted by reading configuration files from storage.

Despite containing many flexible layers of software, most deployed VMs ultimately perform a single function such as acting as a database or Web server. The shift toward single-purpose VMs is a reflection of just how easy it has become to deploy a new virtual computer on demand. Even a decade ago, it would have taken more time and money to deploy a single (physical) machine instance, so the single machine would need to run multiple end-user applications and therefore be carefully configured to isolate the constituent services and users from each other.

The software layers that form a VM haven't yet caught up to this trend, and this represents a real opportunity for optimization—not only in terms of performance by adapting the appliance to its task, but also for improving security by eliminating redundant functionality and reducing the attack surface of services running on the public cloud. Doing so statically is a challenge, however, because of the structure of existing operating systems.

LIMITATIONS OF CURRENT OPERATING SYSTEMS

The modern hypervisor provides a resource abstraction that can be scaled dynamically—both vertically by adding memory and cores, and horizontally by spawning more VMs. Many applications and operating systems can't fully utilize this capability since they were designed before modern hypervisors came about (and the physical analogs such as memory hotplug were never ubiquitous in commodity hardware). Often, external application-level load balancers are added to traditional applications running in VMs in order to make the service respond *elastically* by spawning new VMs when load increases. Traditional systems, however, are not optimized for size or boot time (Windows might apply a number of patches at boot time, for example), so the load balancer must compensate by keeping idle VMs around to deal with load spikes, wasting resources and money.

Why couldn't these problems with operating systems simply be fixed? Modern operating systems are intended to remain resolutely general-purpose to solve problems for a wide audience. For example, Linux runs on an incredibly diverse set of platforms, from low-power mobile devices to high-end servers powering vast data centers. Compromising this flexibility simply to help one class of users improve application performance would not be acceptable.

On the other hand, a specialized server appliance no longer requires an OS to act as a resource multiplexer since the hypervisor can do this at a lower level. One obvious problem with this approach is that most existing code presumes the existence of large but rather calcified interfaces such as POSIX or the Win32 API. Another potential problem is that conventional operating systems provide services such as a TCP/IP stack for communication and a file-system interface for storing persistent data: in our brave new world, where would these come from?

The MirageOS architecture—dubbed *unikernels*—is outlined in figure 1. Unikernels are specialized OS kernels that are written in a high-level language and act as individual software components. A full application (or *appliance*) consists of a set of running unikernels working together as a distributed system. MirageOS is based on the OCaml (http://ocaml.org) language and emits unikernels that run on the Xen hypervisor. To explain how it works, let's look at a radical operating-system architecture from the 1990s that was clearly ahead of its time.

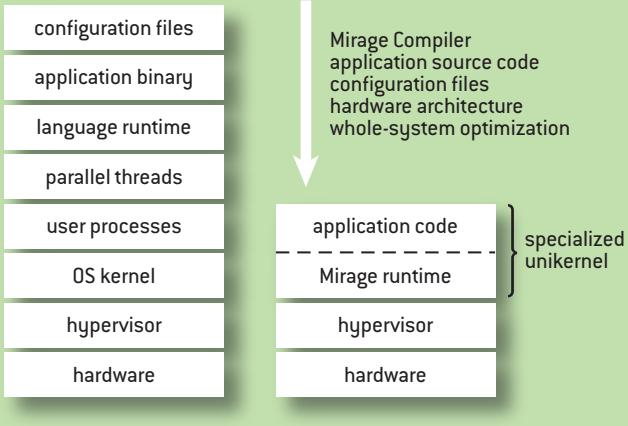
LIBRARY OPERATING SYSTEM

This is not the first time people have asked these existential questions about operating systems. Several research groups have proposed operating-system designs based on an architecture known as a *library operating system* (or libOS). The first such systems were Exokernel[6] and Nemesis[10] in the late 1990s. In a libOS, protection boundaries are pushed to the lowest hardware layers, resulting in: (1) a set of *libraries* that implement *mechanisms*, such as those needed to drive hardware or talk network protocols; and (2) a set of *policies* that enforce access control and isolation in the application layer.

The libOS architecture has several advantages over more conventional designs. For applications where performance—and especially *predictable* performance—is required, a libOS wins by allowing applications to access hardware resources directly without having to make repeated privilege transitions to move data between user space and kernel space. The libOS does not have a central networking service into which both high-priority network packets (such as those from a video conference call) and low-priority packets (such as those from a background file download) are forced to mix and interfere. Instead, libOS applications have entirely separate queues, and packets mix together only when they arrive at the network device.

The libOS architecture has two big drawbacks. First, running multiple applications side by side with strong resource isolation is tricky (although Nemesis did an admirable job of minimizing crosstalk between interactive applications). Second, device drivers must be rewritten to fit the new model. The fast-moving world of commodity PC hardware meant that, no matter how many

FIGURE 1 **Enterprise Component for A Highly Re-configurable Architectural Style**



graduate students were tasked to write drivers, any research libOS prototype was doomed to become obsolete in a few short years. This approach worked only in the realtime operating-system space (e.g., VxWorks) where hardware support is narrower.

Happily, OS virtualization overcomes these drawbacks on commodity hardware. A modern hypervisor provides VMs with CPU time and strongly isolated virtual devices for networking, block storage, USB, and PCI bridges. A libOS running as a VM needs to implement only drivers for these virtual hardware devices and can depend on the hypervisor to drive the real physical hardware. Isolation between libOS applications can be achieved at low cost simply by using the hypervisor to spawn a fresh VM for each distinct application, leaving each VM free to be extremely specialized to its particular purpose. The hypervisor layer imposes a much simpler, less fine-grained policy than a conventional operating system, since it just provides a low-level interface consisting of virtual CPUs and memory pages, rather than the process- and file-oriented architecture found in conventional operating systems.

Although OS virtualization has made the libOS possible without needing an army of device-driver writers, *protocol libraries* are still needed to replace the *services* of a traditional operating system. Modern kernels are all written in C, which excels at low-level programs such as device drivers but lacks the abstraction facilities of higher-level languages and demands careful manual tracking of resources such as memory buffers. As a result, many applications contain memory-handling bugs, which often manifest as serious security vulnerabilities. Other systems researchers have done an admirable job of porting both Windows and Linux to a libOS model,[16] but for us this provided the perfect excuse to explore a less backward-compatible but more naturally integrated high-level language model.

Figure 2 shows the logical workflow in MirageOS. Precise dependency tracking from source code (both local and global libraries) and configuration files lets the full provenance of the deployed kernel binaries be recorded in immutable data stores, sufficient to precisely recompile it on demand.

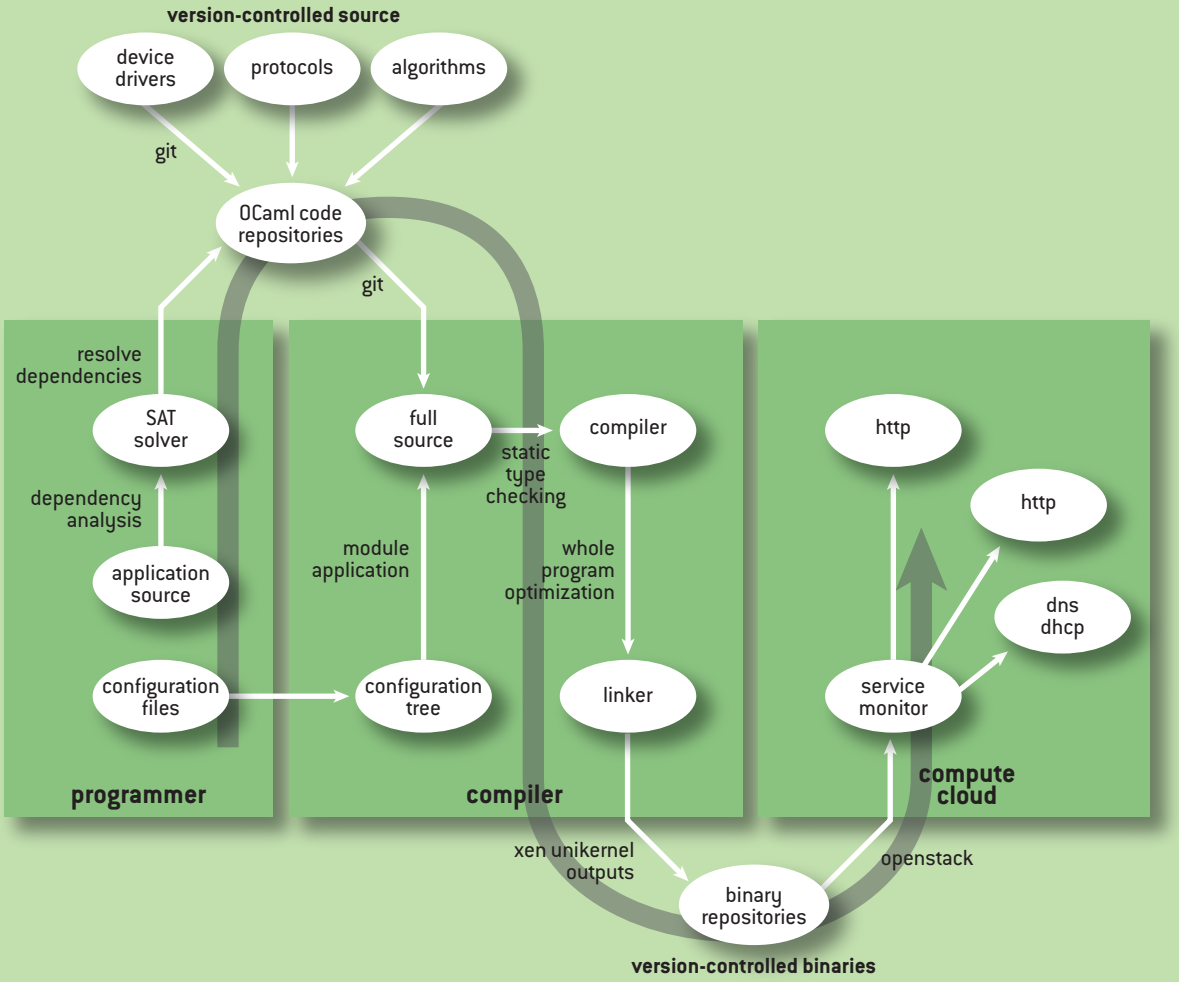
STRONGER PROGRAMMING ABSTRACTIONS

High-level languages are steadily gaining ground in general application development and are increasingly used to glue components together via orchestration frameworks (e.g., Puppet and Chef). Unfortunately, all this logic is typically scattered across software components and is written in several languages. As a result, it is hard to reason statically about the whole system's behavior just by analyzing the source code.

MirageOS aims to unify these diverse interfaces—both kernel and application user spaces—into a single high-level language framework. Some of the benefits of modern programming languages include:

- **Static type checking.** Compilers can classify program variables and functions into types and reject code where a variable of one type is operated on as if it were a different type. Static type checking catches these errors at compile time rather than runtime and provides a flexible way for a systems programmer to protect different parts of a program from each other without depending solely on hardware mechanisms such as virtual memory paging. The most obvious benefit of type checking is the resulting lack of memory errors such as buffer or integer overflows, which are still prevalent in the CERT (Computer Emergency Readiness Team) vulnerability database. A more advanced use is capability-style access control,[19] which can be entirely enforced in a static type system

FIGURE 2 **Logical Workflow in MirageOS**



such as ML's, as long as the code all runs within the same language runtime.

- **Automatic memory management.** Runtime systems relieve programmers of the burden of allocating and freeing memory, while still permitting manual management of buffers (e.g., for efficient I/O). Modern garbage collectors are also designed to minimize application interruptions via incremental and generational collection, thus permitting their use in high-performance systems construction.[7,11]
- **Modules.** When the code base grows, modules partition it into logical components with well-defined interfaces gluing them together. Modules help software development scale as internal implementation details can be abstracted and the scope of a single source-code change can be restricted. Some module systems, such as those found in OCaml and Standard ML, are statically resolved at compilation time and are largely free of runtime costs. The goal is to harness these module systems to build whole systems, crossing traditional kernel and user-space boundaries in one program.

• **Metaprogramming.** If the runtime configuration of a system is partially understood at compile time, then a compiler can optimize the program much more than it would normally be able to. Without knowledge of the runtime configuration, the compiler's hands are tied, as the output program must remain completely generic, just in case. The goal here is to unify configuration and code at compilation time and eliminate waste before deploying to the public cloud.

Together, these features significantly simplify the construction of large-scale systems: managed memory eliminates many resource leaks, type inference results in more succinct source code, static type checking verifies that code matches some abstraction criteria at compilation time rather than execution time, and module systems allow the manipulation of this code at the scales demanded by a full OS and application stack.

A FUNCTIONAL PROTOTYPE IN OCAML

We started building the MirageOS prototype in 2008 with the intention of understanding how far we could unify the programming models underlying library operating systems and cloud-service deployment. The first design decision was to adopt the principles behind *functional programming* to construct the prototype. Functional programming has an emphasis on supporting abstractions that make it easier to track mutability in programs, and previous research has shown that this need not come at the price of performance.[11]

The challenge was to identify the correct modular abstractions to support the expression of an entire operating system and application software stack in a single manageable structure. MirageOS has since grown into a mature set of almost 100 open-source libraries that implement a wide array of functionality, and it is starting to be integrated into commercial products such as Citrix XenServer.[17]

Figure 2 illustrates MirageOS's design. It grants the compiler a much broader view of source-code dependencies than a conventional cloud deployment cycle:

• All source-code dependencies of the input application are explicitly tracked, including all the libraries required to implement kernel functionality. MirageOS includes a build system that internally uses a SAT solver (using the OPAM package manager, with solvers from the Mancoosi project) to search for compatible module implementations from a published online package set. Any mismatches in interfaces are caught at compile time because of OCaml's static type checking.

• The compiler can then output a full stand-alone kernel instead of just a Unix executable. These unikernels are single-purpose libOS VMs that perform only the task defined in their application source and configuration files, and they depend on the hypervisor to provide resource multiplexing and isolation. Even the bootloader, which has to set up the virtual memory page tables and initialize the language runtime, is written as a simple library. Each application links to the specific set of libraries it needs and can glue them together in application-specific ways.

• The specialized unikernels are deployed on the public cloud. They have a significantly smaller attack surface than the conventional virtualized equivalents and are more resource-efficient in terms of boot time, binary size, and runtime performance.

WHY OCAML?

OCaml is the sole base language for MirageOS for a few key reasons. It is a full-fledged systems programming language with a flexible programming model that supports functional, imperative,

and object-oriented styles within a single, ML-inspired type system. It also features a portable single-threaded runtime that makes it ideal for porting to restricted environments such as a barebones Xen VM. The compiler heavily emphasizes static type checking, and the resulting binaries are fast native code with minimal runtime type information. Principal type inference allows type annotations to be safely omitted, and the module system is among the most powerful in a general-purpose programming language in terms of permitting flexible and safe code reuse and refactoring. Finally, there were several examples of large-scale uses of OCaml in industry[14] and within Xen itself,[17] and the positive results were encouraging before embarking on the large multiyear project that MirageOS turned out to be.

MODULAR OPERATING-SYSTEM LIBRARIES

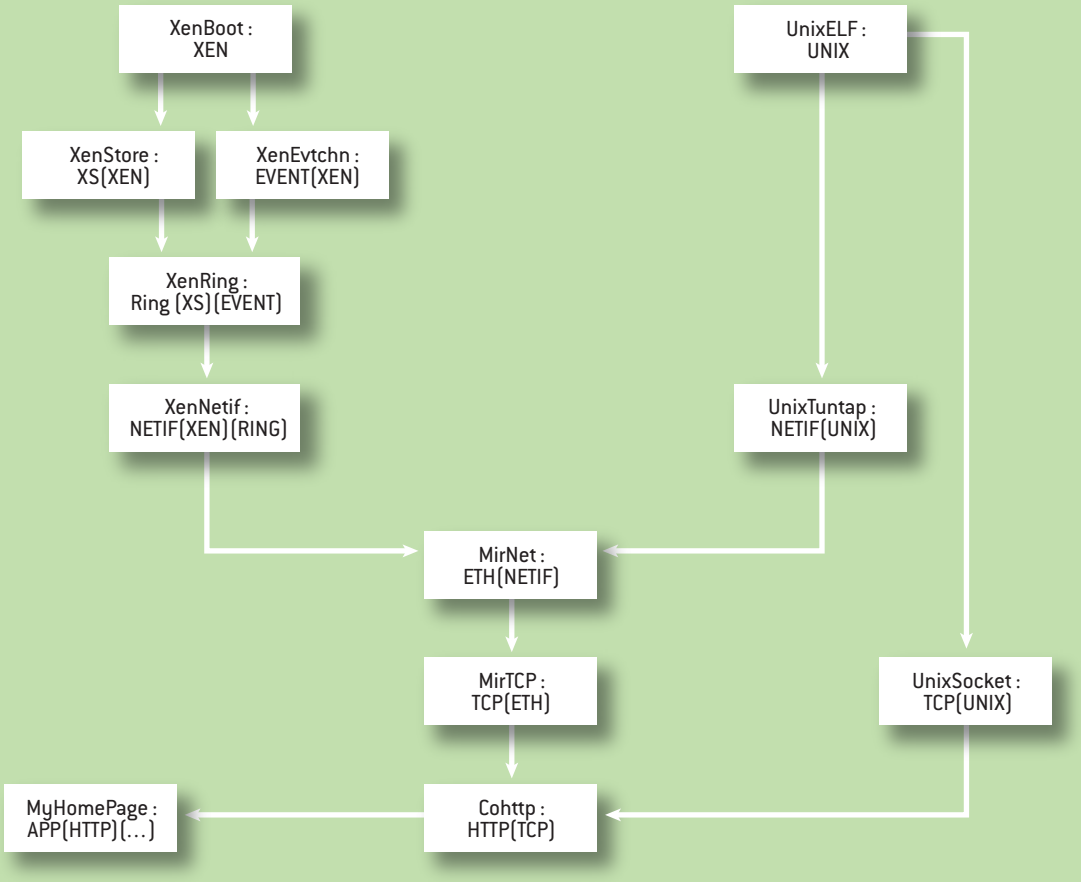
OCaml supports the definition of *module signatures* (a collection of data-type and function declarations) that abstract the implementation of *module structures* (definitions of concrete data types and functions). Modules can be parameterized over signatures, creating *functors* that define operations across other data types. (For more information about OCaml modules, functors, and objects, see *Real World OCaml*, published by O'Reilly and available at https://realworldocaml.org.) We applied the OCaml module system to breaking the usually monolithic OS kernel functionality into discrete units. This lets programmers build code that can be *progressively specialized* as it is being written, starting from a process in a familiar Unix environment and ending up with a specialized cloud unikernel running on Xen.

Consider a simple example. Figure 3 shows a partial module graph for a static Web server. Libraries are module graphs that abstract over operating-system functionality, and the OPAM package manager solves constraints over the target architecture. The application `MyHomePage` depends on an `HTTP` signature that is provided by the `Cohttp` library. Developers just starting out want to explore their code interactively using a Unix-style development environment. The `Cohttp` library needs a TCP implementation to satisfy its module signature, which can be provided by the `UnixSocket` library.

When the programmers are satisfied that their HTTP logic is working, they can recompile to switch away from using Unix sockets to the OCaml TCP/IP stack shown by `MirTCP` in figure 3. This still requires a Unix kernel but only as a shell to deliver Ethernet frames to the Web-server process (which now incorporates an OCaml TCP/IP stack as part of the application). The last compilation strategy drops the dependency on Unix entirely and recompiles the `MirNet` module to link directly to a Xen network driver, which in turn pulls in all the dependencies it needs to boot on Xen. This progressive recompilation is key to the usability of MirageOS, since we can evolve from the tried-and-tested Linux or FreeBSD functionality gradually but still end up with specialized unikernels that can be deployed on the public cloud. This modular operating-system structure has led to a number of other back ends being implemented in a similar vein to Xen. MirageOS now has experimental back ends that implement a simulator in NS3 (for large-scale functional testing), a FreeBSD kernel module back end, and even a JavaScript target by using the `js_of_ocaml` compiler. A natural consequence of this modularity is that it is easier to write portable code that defines exactly what it needs from a target platform, which is increasingly difficult on modern operating systems with the lack of a modern equivalent of POSIX (which has led Linux, FreeBSD, Mac OS X, and Windows to have numerous incompatible APIs for high-performance services).

FIGURE 3

A Partial Module Graph for A Static Web Server



CONFIGURATION AND STATE

Libraries in MirageOS are designed in as functional a style as possible: they are reentrant with explicit state handles, which are in turn serializable so that they can be reconstructed explicitly. An application consists of a set of libraries plus some configuration code, all linked together. The configuration is structured as a tree roughly like a file system, with subdirectories being parsed by each library to initialize their own values (reminiscent of the Plan 9 operating system). All of this is connected by *metaprogramming*—an OCaml program generates more OCaml code that is compiled until the desired target is reached.

The metaprogramming extends into storage as well. If an application uses a small set of files (which would normally require all the baggage of block devices and a file system), MirageOS can convert it into a static OCaml module that satisfies the file-system module signature, relieving it of the need for an external storage dependency. The entire MirageOS home page (http://openmirage.org) is served in this manner.

One (deliberate) consequence of metaprogramming is that large blocks of functionality may be entirely missing from the output binary. This makes dynamic reconfiguration of the most

specialized targets impossible, and a configuration change requires the unikernel to be relinked. The lines of active (i.e., post-configuration) code involved in a MirageOS Web server are shown in table 1, giving a sense of the small amount of code involved in such a recompilation.

LINKING THE XEN UNIKERNEL

In a conventional OS, application source code is first compiled into object files via a native-code compiler and then handed off to a linker that generates an executable binary. After compilation, a dynamic linker loads the executable and any shared libraries into a *process* with its own address space. The process can then communicate with the outside world by system calls, mediated by the operating-system kernel. Within the kernel, various subsystems such as the network stack or virtual memory system process system calls and interact with the hardware.

In MirageOS, the OCaml compiler receives the source code for an entire kernel's worth of code and links it into a stand-alone native-code object file. It is linked against a minimal runtime that provides boot support and the garbage collector. There is no preemptive threading, and the kernel is event-driven via an I/O loop that polls Xen devices.

The Xen unikernel compilation derives its performance benefit from the fact that the running kernel has a single virtual address space, designed to run only the OCaml runtime. The virtual address space of the MirageOS Xen unikernel target is shown in figure 4. Since all configuration information is explicitly part of the compilation, there is no longer a need for the usual dynamic linking support that requires executable mappings to be added after the VM has booted.[13]

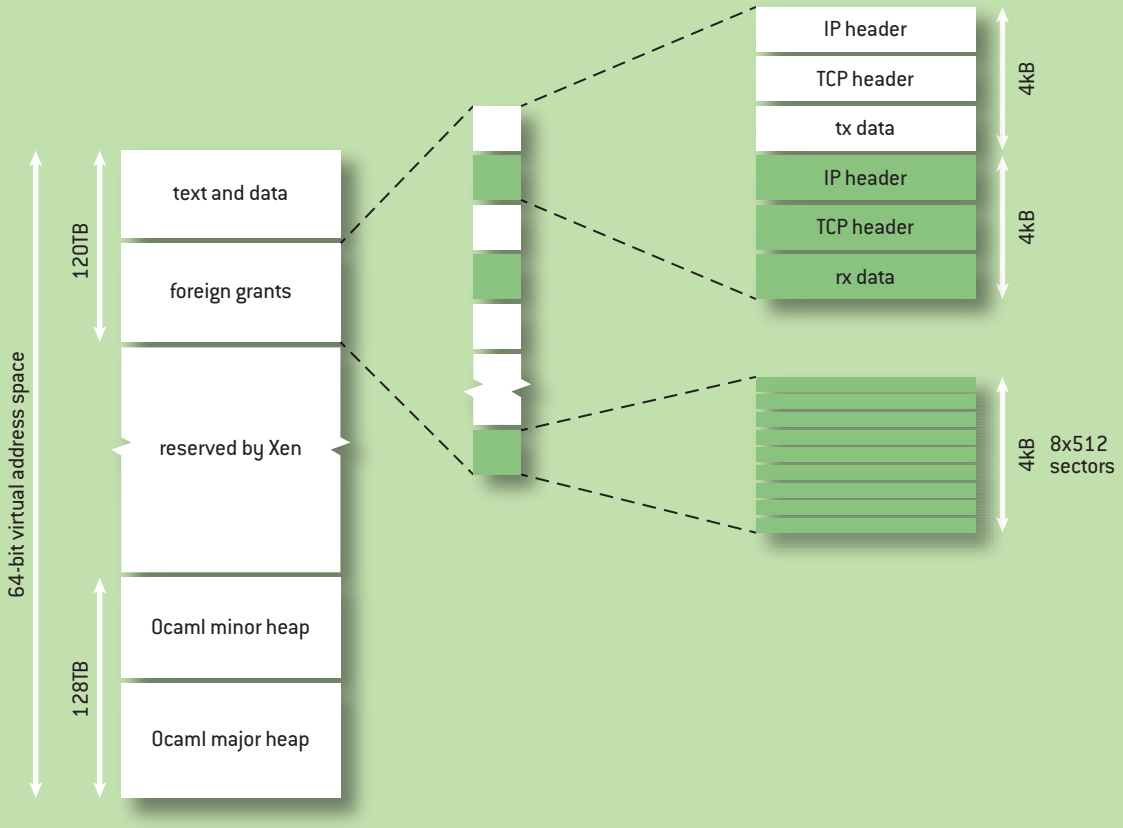
BENEFITS

Consider the life cycle of a traditional application. First the source code is compiled to a binary. Later, the binary is loaded into memory and an OS process is created to execute it. The first thing the running process will do is read its configuration file and specialize itself to the environment it finds itself in. Many different applications will run exactly the same binary, obtained from the same binary package, but with different configuration files. These configuration files are effectively additional program code, except they're normally written in ad-hoc languages and interpreted at runtime rather than compiled.

TABLE 1 **Approximate sizes of libraries used by a typical MirageOS Xen unikernel running a Web server**

Library	C/kLOC	OCaml/kLOC
Boot	18	0
OCaml runtime	20	0
threads	5	27
interdomain comms	trace	1
network driver	0	1
TCP/IP	trace	12
block driver	0	1
HTTP	0	11
Total	43	52

FIGURE 4 **Virtual Address Space of the MirageOS Xen Unikernel Target**



DEPLOYMENT AND MANAGEMENT

Configuration is a considerable overhead in managing the deployment of a large cloud-hosted service. The traditional split between the *compiled* (code) and *interpreted* (configuration) is unnecessary with unikernel compilation. Application configuration is code—perhaps in an embedded domain-specific language—and the compiler can analyze and optimize across the whole unikernel.

In MirageOS, rather than treating the database, web server, etc. as independent applications that must be connected by configuration files, they are treated as libraries within a single application, allowing the application developer to configure them using either simple library calls for dynamic parameters or metaprogramming tools for static parameters. This has the useful effect of making configuration decisions explicit and programmable in a host language rather than manipulating many ad-hoc text files and thus benefiting from static-analysis tools and the compiler's type checker. The result is a big reduction in the effort needed to configure complex multiservice application VMs.

One downside to a unikernel is the burden it places on the cloud orchestration layers because of

the need to schedule many more VMs with greater churn (since every reconfiguration requires the VM to be redeployed). The popular orchestration implementations have grown rather organically in recent years and consist of many distributed components that are not only difficult to manage, but also relatively high in latency and resource consumption.

One of the first production uses for MirageOS is to fix the cloud-management stacks by evolving the OCaml code within XenServer[17] toward the structured unikernel worldview. This turns the monolithic management layer into a more agile set of intercommunicating VMs that can be scheduled and restarted independently. MirageOS makes constructing these single-purpose VMs easy: they are first built and tested as regular Unix applications before flipping a switch and relinking against the Xen kernel libraries (http://openmirage.org/blog/xenstore-stub-domain). When they are combined with Xen driver domains,[3] they can dramatically increase the security and robustness of the cloud-management stack.

RESOURCE EFFICIENCY AND CUSTOMIZATION

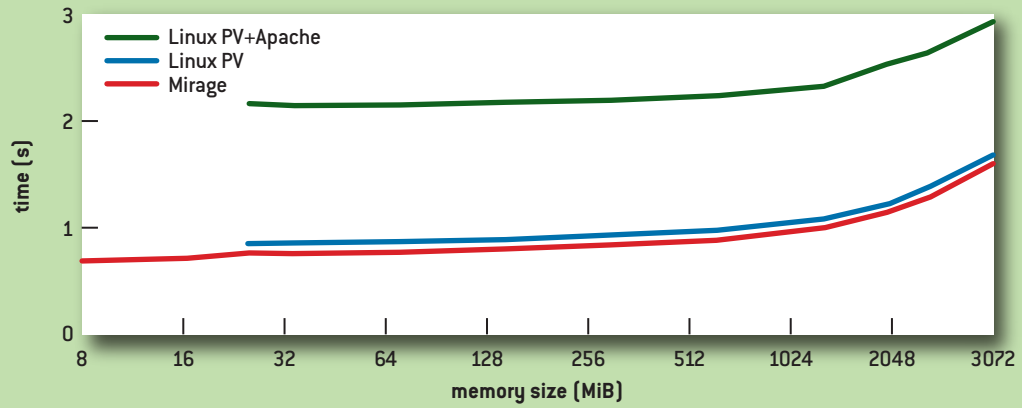
The cloud is an environment where all resource usage is metered and rented. At the same time, multitenant services suffer from variability in load that encourages rapid scaling of deployments—both *up* to meet current demand and *down* to avoid wasting money. In MirageOS, features that are not used in a particular build are not included, and whole-system optimization techniques can be used to eliminate waste at compilation time rather than deployment time. In the most specialized mode, all configuration files are statically evaluated, enabling extensive dead-code elimination at the cost of having to recompile to reconfigure the service.

The small binary size of the unikernels (on the order of hundreds of kilobytes in many cases) makes deployment to remote data centers across the Internet much smoother. Boot time is also easily less than a second, making it feasible to boot a unikernel in response to incoming network packets.

Figure 5 compares the boot time of a service in MirageOS and a Linux/Apache distribution. The boot time of a stripped-down Linux kernel and MirageOS are similar, but the inefficiency creeps

FIGURE 5

Boot Time



into Linux as soon as it has to initialize the user-space applications. The MirageOS unikernel is ready to serve traffic as soon as it boots.

The MLton[20] compiler pioneered WPO (whole program optimization), where an application and all of its libraries are optimized together. In the libOS world, a whole program is actually a whole operating system: this technique can now optimize all the way from application-level code to low-level device drivers. Traditional systems eschew WPO in favor of dynamic linking, sometimes in combination with JIT (just-in-time) compiling, where a program is analyzed dynamically, and optimized code is generated on the fly. Whole-program, compile-time optimization is more appropriate for cloud applications that care about resource efficiency and reducing their attack surface. Other research elaborates on the security benefits.[13]

An interesting recent trend is a move toward operating-system *containers* in which each container is managed by the same operating-system kernel but with an isolated file system, network, and process group. Containers are quick to create since there is no need to boot a new kernel, and they are fully compatible with existing kernel interfaces. However, these gains are made at the cost of reduced security and isolation; unikernels share only the minimal hypervisor services via a small API, which is easy to understand and audit. Unikernels demonstrate that layering language runtimes onto a hypervisor is a viable alternative to lightweight containers.

A NEW FRONTIER OF PORTABILITY

The structure of MirageOS libraries shown in figure 3 explicitly encodes what the library needs from its execution environment. While this has conventionally meant a Posix-like kernel and user space, it is now possible to compile OCaml into more foreign environments, including FreeBSD kernel modules, JavaScript running in the browser, or (as the Scala language does) directly targeting the JVM (Java Virtual Machine). Some care is still required for execution properties that are not abstractable in the OCaml type system.

For example, floating-point numbers are generally forbidden when running as a kernel module; thus, a modified compiler emits a type error if floating-point code is used when compiling for that hardware target.

Other third-party OCaml code often exhibits a similar structure, making it much easier to work under MirageOS. For example, Arakoon (http://arakoon.org) is a distributed key-value store that implements an efficient multi-Paxos consensus algorithm. The source-code patch to compile it under MirageOS touched just two files and was restricted to adding a new module definition that mapped the Arakoon back-end storage to the Xen block driver interface.

UNIKERNELS IN THE WILD

MirageOS is certainly not the only unikernel that has emerged in the past few years, although it is perhaps the most extreme in terms of exploring the clean-slate design space. Table 2 shows some of the other systems that build unikernels. HalVM[8] is the closest to the MirageOS philosophy, but it is based on the famously pure and lazy Haskell language rather than the strictly evaluated OCaml. On the other end of the spectrum, OSv[2] and rump kernels[9] provide a compatibility layer for existing applications, and deemphasize the programming model improvements and type safety that guide MirageOS. The Drawbridge project[16] converts Windows into a libOS with just a reported 16-MB

TABLE 2 **Other unikernel implementations**

Unikernel	Language	Targets
Mirage[13]	OCaml	Xen, kFreeBSD, POSIX, WWW/js
Drawbridge[16]	C	Windows "picoprocess"
HaIVM[8]	Haskell	Xen
ErlangOnXen	Erlang	Xen
OSv[2]	C/Java	Xen, KVM
GUK	Java	Xen
NetBSD "rump"[9]	C	Xen, Linux kernel, POSIX
ClickOS [14]	C++	Xen

overhead per application, but it exposes higher-level interfaces than Xen (such as threads and I/O streams) to gain this efficiency.

Ultimately, the public cloud should support all these emerging projects as first-class citizens just as Linux and Windows are today. The Xen Project aims to support a brave new world of *dust clouds*: tiny one-shot VMs that run on hypervisors with far greater density than is currently possible and that self-scale their resource needs by constantly calling into the cloud fabric. The libOS principles underlying MirageOS mean that it is not limited to running on a hypervisor platform—many of the libraries can be compiled to multiscale environments,[12] ranging from ARM smartphones to bare-metal kernel modules. To understand the implications of this flexibility, we've been exploring use-cases ranging from managing personal data[4] and facilitating anonymous communication,[15] to building software-defined data-center infrastructure.

ACKNOWLEDGMENTS

The MirageOS effort has been a large one and would not be possible without the intellectual and financial support of several sources. The core team of Richard Mortier, Thomas Gazagnaire, Jonatham Ludlam, Haris Rotsos, Balraj Singh, and Vincent Bernardoff have toiled to help us build the clean-slate OCaml code, with constant support and feedback from Jon Crowcroft, Steven Hand, Ian Leslie, Derek McAuley, Yaron Minsky, Andrew Moore, Simon Moore, Alan Mycroft, Peter G. Neumann, and Robert N. M. Watson. The wider OCaml community has contributed thousands of OPAM packages that greatly increase the utility of MirageOS, and Pierre Chambart and Fabrice Le Fessant from OCamlPro integrated OCaml compiler optimizations that sped up MirageOS/Xen. Raphael Proust and Gabor Pali contributed the JavaScript and kFreeBSD targets on summer internships. David Chisnall, Tim Deegan, Marius Eriksen, Nate Foster, Arjun Guha, Tim Harris, Alex Ho, Jon Howell, Stephen Kell, Timothy G. Griffin, Tom Kelly, Ewan Mellor, Prashanth Mundkur, Derek G. Murray, Fernando Ramos, Malte Schwarzkopf, Peter Sewell, David Sheets, Ripduman Sohan, Leo White, and Jeremy Yallop have given feedback on several iterations of this work. We have benefited from significant open source code contributions from Citrix Systems, Jane Street, Lexifi, and, of course, the OCaml development team at the Galium group at INRIA led by Xavier Leroy. The Linux Foundation also accepted MirageOS as a Xen Incubator Project, thanks to the efforts of Lars Kurth and Amir Chaudhry.

This work was primarily supported by Horizon Digital Economy Research, RCUK grant EP/G065802/1. A portion was sponsored by DARPA (Defense Advanced Research Projects Agency) and AFRL (Air Force Research Laboratory), under contract FA8750-11-C-0249. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

MirageOS is available freely at http://openmirage.org. We welcome feedback, patches, and improbable stunts using it.

REFERENCES

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*: 164-177.
2. Cloudius Systems. OSv; https://github.com/cloudius-systems/osv.
3. Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., Warfield, A. 2011. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*: 189-202.
4. Crowcroft, J., Madhavapeddy, A., Schwarzkopf, M., Hong, T., Mortier, R. Unclouded vision. In *Proceedings of the International Conference on Distributed Computing and Networking (ICDCN)* 29-40.
5. Eisenstadt, M. My hairiest bug war stories. 1997. *Communications of the ACM* 40(4): 30-37.
6. Engler, D. R., Kaashoek, M. F., O'Toole, Jr., J. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*: 251-266.
7. Eriksen, M. 2013. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS):* 5:1-5:7.
8. Galois Inc. The Haskell Lightweight Virtual Machine (HaLVM) source archive; https://github.com/GaloisInc/HaLVM.
9. Kantee, A. 2012. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. Ph.D. thesis, Aalto University, Espoo, Finland.
10. Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., Hyden, E. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications* 14(7): 1280-1297.
11. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R. 2007. Melange: creating a "functional" Internet. *SIGOPS Operating Systems Review* 41(3): 101-114.
12. Madhavapeddy, A., Mortier, R., Crowcroft, J., S. Hand, S. 2010. Multiscale not multicore: efficient heterogeneous cloud computing. In *Proceedings of ACM-BCS Visions of Computer Science*. Electronic Workshops in Computing, Edinburgh, UK.
13. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS):* 461-472.
14. Minsky, Y. 2011. OCaml for the masses. *Communications of the ACM* 54(11): 53-58.
15. Mortier, R., Madhavapeddy, A., Hong, T., Murray, D., Schwarzkopf, M. 2010. Using dust clouds to

enhance anonymous communication. In *Proceedings of the 18th International Workshop on Security Protocols (IWSP).*

16. Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R., Hunt, G. C. 2011. 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)*: 291-304.
17. Scott, D., Sharp, R., Gazagnaire, T., Madhavapeddy, A. 2010. Using functional programming within an industrial product group: perspectives and perceptions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP):* 87-92.
18. Vinge, V. 1992. *A Fire Upon the Deep*. New York, NY: Tor Books.
19. Watson, R. N. M. 2013. A decade of OS access-control extensibility. *Communications of the ACM* 56(2): 52-63.
20. Weeks, S. 2006. Whole-program compilation in MLton. In *Proceedings of the 2006 Workshop on ML*.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ANIL MADHAVAPEDDY is a Senior Research Fellow at the University of Cambridge, based in the Systems Research Group. He was on the original team that developed the Xen hypervisor and the industry-leading XenServer management toolstack written in OCaml. XenServer has been deployed on millions of hosts and drives critical infrastructure for many Fortune 500 companies. Prior to obtaining his PhD in 2006 from the University of Cambridge, Anil had a diverse background in industry at Network Appliance, NASA, and Internet Vision. He is an active member of the open-source community with contributions to OpenBSD and OCaml, is on the steering committee of the Commercial Uses of Functional Programming workshop, and is an author of *Real World OCaml* from O'Reilly.

DAVE SCOTT is a Principal Architect at Citrix Systems where he works on the XenServer virtualization platform. Dave's main focus is on improving XenServer reliability and performance through exploiting advances in open-source software and high-level languages. Previously Dave was a member of technical staff at Fraser Research in Princeton, NJ. Dave holds a PhD in computer science from the University of Cambridge.