Unikernels: Library Operating Systems for the Cloud

Anil Madhavapeddy, Richard Mortier[1], Charalampos Rotsos, David Scott[2], Balraj Singh,
Thomas Gazagnaire[3], Steven Smith, Steven Hand and Jon Crowcroft

University of Cambridge, University of Nottingham[1], Citrix Systems Ltd[2], OCamlPro SAS[3]
first.last@cl.cam.ac.uk, first.last@nottingham.ac.uk, dave.scott@citrix.com, first@ocamlpro.com

Abstract

We present *unikernels*, a new approach to deploying cloud services via applications written in high-level source code. Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform. In return they offer significant reduction in image sizes, improved efficiency and security, and should reduce operational costs. Our Mirage prototype compiles OCaml code into unikernels that run on commodity clouds and offer an order of magnitude reduction in code size without significant performance penalty. The architecture combines static type-safety with a single address-space layout that can be made immutable via a hypervisor extension. Mirage contributes a suite of type-safe protocol libraries, and our results demonstrate that the hypervisor is a platform that overcomes the hardware compatibility issues that have made past library operating systems impractical to deploy in the real-world.

Categories and Subject Descriptors D.4 [*Operating Systems*]: Organization and Design; D.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Experimentation, Performance

1. Introduction

Operating system virtualization has revolutionised the economics of large-scale computing by providing a platform on which customers rent resources to host virtual machines (VMs). Each VM presents as a self-contained computer, booting a standard OS kernel and running unmodified application processes. Each VM is usually specialised to a particular role, e.g., a database, a webserver, and scaling out involves cloning VMs from a template image.

Despite this shift from applications running on multi-user operating systems to provisioning many instances of single-purpose VMs, there is little actual specialisation that occurs in the image that is deployed to the cloud. We take an extreme position on specialisation, treating the final VM image as a single-purpose appliance rather than a general-purpose system by stripping away functionality at compile-time. Specifically, our contributions are: (*i*) the *unikernel* approach to providing sealed single-purpose appliances, particularly suitable for providing cloud services; (*ii*) evaluation of a complete implementation of these techniques using a functional programming language (OCaml), showing that the benefits of type-safety need not damage performance; and (*iii*) libraries and language extensions supporting systems programming in OCaml.

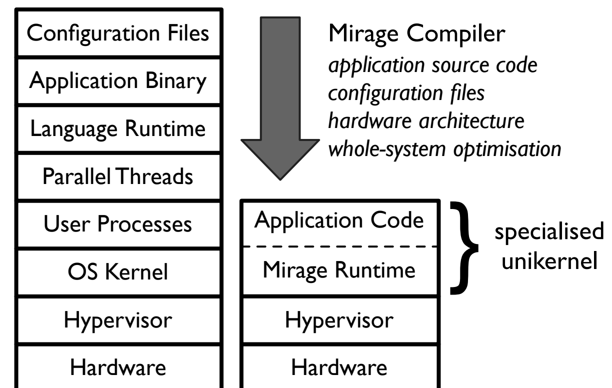


Figure 1: Contrasting software layers in existing VM appliances *vs.* unikernel's standalone kernel compilation approach.

The unikernel approach builds on past work in library OSs [1–3]. The entire software stack of system libraries, language runtime, and applications is compiled into a single bootable VM image that runs directly on a standard hypervisor (Figure 1). By targeting a standard hypervisor, unikernels avoid the hardware compatibility problems encountered by traditional library OSs such as Exokernel [1] and Nemesis [2]. By eschewing backward compatibility, in contrast to Drawbridge [3], unikernels address cloud services rather than desktop applications. By targeting the commodity cloud with a library OS, unikernels can provide greater performance and improved security compared to Singularity [4]. Finally, in contrast to Libra [5] which provides a libOS abstraction for the JVM over Xen but relies on a separate Linux VM instance to provide networking and storage, unikernels are more highly-specialised single-purpose appliance VMs that directly integrate communication protocols.

We describe a complete unikernel prototype in the form of our OCaml-based Mirage implementation (§3). We evaluate it via micro-benchmarks and appliances providing DNS, OpenFlow, and HTTP (§4). We find sacrificing source-level backward compatibility allows us to increase performance while significantly improving the security of external-facing cloud services. We retain compatibility with external systems via standard network protocols such as TCP/IP, rather than attempting to support POSIX or other conventional standards for application construction. For example, the Mirage DNS server outperforms both BIND 9 (by 45%) and the high-performance NSD server (§4.2), while using very much smaller VM images: our unikernel appliance image was just 200 kB while the BIND appliance was over 400 MB. We conclude by discussing our experiences building Mirage and its position within the state of the art (§5), and concluding (§6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03. . . $15.00

2. Architecture of an Appliance

Virtualisation is the enabling technology for the cloud, widely deployed via hypervisors such as Xen [6]. VM appliances are built to provide a small, fixed set of services. Thus, datacenter appliances typically consist of a Linux or Windows VM booted over Xen with loosely-coupled components: a guest OS kernel hosting a primary application (e.g., MySQL, Apache) with other services (e.g., cron, NTP) running in parallel; and typically attaching an external storage device with configuration files and data.

Our key insight is that the hypervisor provides a virtual hardware abstraction that can be scaled dynamically – both vertically by adding memory and vCPUs, and horizontally by spawning more VMs. This provides an excellent target for *library operating systems* (libOSs), an old idea [1, 2] recently revisited to break up monolithic OSs [3]. LibOSs have never been widely deployed due to the difficulty of supporting a sufficient range of real-world hardware, but deploying on a hypervisor such as Xen allows us to bypass this issue by using the hypervisor's device drivers, affording the opportunity to build a practical, clean-slate libOS that runs natively on cloud computing infrastructure.

We dub these VMs *unikernels*: specialised, sealed, single-purpose libOS VMs that run directly on the hypervisor. A libOS is structured very differently from a conventional OS: all services, from the scheduler to the device drivers to the network stack, are implemented as libraries linked directly with the application. Coupled with the choice of a modern statically type-safe language for implementation, this affords configuration, performance and security benefits to unikernels.

2.1 Configuration and Deployment

Configuration is a considerable overhead in managing the deployment of a large cloud-hosted service. Although there are (multiple) standards for location and format of configuration files on Linux, and Windows has the Registry and Active Directory, there are no standards for many aspects of application configuration. To address this, for example, Linux distributions typically resort to extensive shell scripting to glue packages together.

Unikernels take a different approach, by integrating configuration into the compilation process. Rather than treating the database, web server, etc., as independent applications which must be connected together by configuration files, unikernels treat them as libraries within a single application, allowing the application developer to configure them using either simple library calls for dynamic parameters, or build system tools for static parameters. This has the useful effect of making configuration decisions explicit and programmable in a host language rather than manipulating many adhoc text files, and hence benefiting from static analysis tools and the compiler's type-checker. The end result is a big reduction in the effort needed to configure complex multi-service application VMs.

2.2 Compactness and Optimisation

Resources in the cloud are rented, and minimising their use reduces costs. At the same time, multi-tenant services suffer from high variability in load that incentivises rapid scaling of deployments to meet current demand without wasting money. Unikernels link libraries that would normally be provided by the host OS, allowing the Unikernel tools to produce highly compact binaries via the normal linking mechanism. Features that are not used in a particular compilation are not included and whole-system optimization techniques can be used. In the most specialised 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 (on the order of kilobytes in many cases) makes deployment to remote datacenters across the Internet much smoother.

2.3 Unikernel Threat Model and Implications

Before considering the security implications of the unikernel abstraction, we first state our context and threat model. We are concerned with software that provides network-facing services in multi-tenant datacenters. Customers of a cloud provider typically must trust the provider not to be malicious. However, software running in such an environment is under constant threat of attack, from both other tenants and Internet-connected hosts more generally.

Unikernels run above a hypervisor layer and treat it and the control domain as part of the trusted computing base (for now, see §5.3). However, rather than adopt a multi-user access control mechanism that is inordinately complex for a specialised appliance, unikernels use the hypervisor as the sole unit of isolation and let applications trust external entities via protocol libraries such as SSL or SSH. Internally, unikernels adopt a defence in depth approach: firstly by compile-time specialisation, then by pervasive type-safety in the running code, and finally via hypervisor and toolchain extensions to protect against unforseen compiler or runtime bugs.

2.3.1 Single Image Appliances

The usual need for backwards compatibility with existing applications, e.g., the POSIX API, the OS kernel and the many userspace binaries involved mean that even the simplest appliance VM contains several hundred thousand, if not millions of, lines of active code that must be executed every time it boots (§4.5). Even widely deployed codebases such as Samba and OpenSSL still contain remote code execution exploits published as recently as April 2012 [7, 8], and serious data leaks have become all too commonplace in modern Internet services. A particularly insidious problem is that misconfiguring an image can leave unnecessary services running that can significantly increase the remote attack surface.

A unikernel toolchain performs as much compile-time work as possible to eliminate unnecessary features from the final VM. All network services are available as libraries, so only modules explicitly referenced in configuration files are linked in the output. The module dependency graph can be easily statically verified to only contain the desired services. While there are some Linux package managers that take this approach [9], they are ultimately constrained by having to support dynamic POSIX applications.

The trade-off with using too many static configuration directives that are compiled into the image is that VMs can no longer be cloned by taking a copy-on-write snapshot of an existing image. If this is required, a dynamic configuration directive can be used (e.g., DHCP instead of a static IP). Our prototype Mirage unikernels contain substantially fewer lines of code than the Linux equivalent, and the resulting images are significantly smaller (§4.5).

2.3.2 Pervasive Type-Safety

The requirement to be robust against remote attack strongly motivates use of a type-safe language. An important decision is whether to support multiple languages within the same unikernel. An argument for multiple languages is to improve backwards compatibility with existing code, but at the cost of increasing the complexity of a single-image system and dealing with interoperability between multiple language runtimes.

The alternative is to eschew source-level compatibility and rewrite system components entirely in one language and specialise that toolchain as best as possible. Although it is a daunting engineering challenge to rewrite protocols such as TCP, this is possible for an experimental system such as our Mirage prototype. In choosing this path, we support interoperability at the *network protocol* level: components communicate using type-safe, efficient implementations of standard network protocols. The advantage of running on a hypervisor is that the reverse is also possible: existing non-OCaml code can be encapsulated in separate VMs and

communicated with via message-passing, analogous to processes in a conventional OS (§5.2). Likewise, access control within the appliance no longer requires userspace processes, instead depending on the language's type-safety to enforce restrictions. the virtual address space can be simplified into a single-address space model.

Mirage's single-language focus eases the integration of security techniques to protect the remaining non-type-safe components of the system (notably, the garbage collector) and to provide defence-in-depth in case a compiler bug allows the type-safety property to be violated. Some of these, such as stack canaries and guard pages, are straightforward translations of standard techniques and so we do not discuss them further. However, two depend on the unique properties of the unikernel environment and we describe these next.

2.3.3 Sealing and VM Privilege Dropping

As unikernels are single-image and single-address space, they exercise fewer aspects of the VM interface and can be sealed [10] at runtime to further defend against bugs in the runtime or compiler. This means that any code not present in the unikernel at compile time will never be run, completely preventing code injection attacks. Implementing this policy is very simple: as part of its start-of-day initialisation, the unikernel establishes a set of page tables in which no page is both writable and executable and then issues a special *seal* hypercall which prevents further page table modifications. The memory access policy in effect when the VM is sealed will be preserved until it terminates. The hypervisor changes necessary to implement the sealing operation are themselves very simple;[1] by contrast, implementing an equivalent *Write Xor Execute* [11] policy in a conventional operating system requires extensive modifications to libraries, runtimes, and the OS kernel itself.

This approach does mean that a running VM cannot expand its heap, but must instead pre-allocate all the memory it needs at startup (allocation *within* the heap is unaffected, and the hypervisor can still overcommit memory between VMs). This is a reasonable constraint on cloud infrastructures, where the memory allocated to the VM has already been purchased. The prohibition on page table modification does not apply to I/O mappings, provided that they are themselves non-executable and do not replace any existing data, code, or guard pages. This means that I/O is unaffected by sealing a VM, and does not inherently invalidate the memory access policy.

This optional facility is the only element of unikernels that requires a patch to the hypervisor instead of running purely in the guest. The privilege dropping patch is very simple and would benefit any other single-address space operating system, and so it is being upstreamed to the main Xen distribution. Note that Mirage can run on unmodified versions of Xen without this patch, albeit losing this layer of the defence-in-depth security protections.

2.3.4 Compile-Time Address Space Randomization

While VM sealing prevents an attacker from introducing attack code, it does not prevent them from executing code which is already present. Although use of whole-system optimization can eliminate many targets for such an attack, enough might be left to assemble a viable exploit using return-oriented programming style techniques. Conventionally, these would be protected against using runtime address space randomization, but this requires runtime linker code that would introduce significant complexity into the running unikernel. Fortunately, it is also unnecessary. The unikernel model means that reconfiguring an appliance means recompiling it, potentially for every deployment. We can thus perform address space randomisation at compile time using a freshly generated linker script, without impeding any compiler optimisations and without adding any runtime complexity.

[1] Including the API definition, our patch to Xen 4.1 added fewer than 50 lines of code in total.

3. Mirage Unikernels

Our Mirage prototype produces unikernels by compiling and linking OCaml code into a bootable Xen VM image. We implement all but the lowest-level features in OCaml and, to assist developers testing and debugging their code, provide the ability to produce POSIX binaries that run Mirage services on UNIX, as well as Xen VM images. We now discuss some of the key design decisions and components of Mirage: use of OCaml (§3.1), the PVBoot library that initialises a basic environment (§3.2), a modified language runtime library for heap management and concurrency (§3.3), and its type-safe device drivers (§3.4) and I/O stack (§3.5).

3.1 Why OCaml?

We chose to implement Mirage in OCaml for four key reasons. First, OCaml is a full-fledged systems programming language [12] with a flexible programming model that supports functional, imperative and object-oriented programming, and its brevity reduces lines-of-code (LoC) counts that are often considered correlated with attack surface. Second, OCaml has a simple yet high-performance runtime making it an ideal platform for experimenting with the unikernel abstraction that interfaces the runtime with Xen. Third, its implementation of static typing eliminates type information at compile-time while retaining all the benefits of type-safety, another example of specialisation. Finally, the open-source Xen Cloud Platform [12] and critical system components [13, 14] are implemented in OCaml, making integration straightforward.

However, this choice does impose tradeoffs. OCaml is still a relatively esoteric language compared with other systems languages such as C/C++. Using OCaml also necessitated a significant engineering effort to rebuild system components, particularly the storage and networking stacks. Given the other benefits of OCaml, we do not feel either of these are significant impediments for a research prototype. One early decision decision we took is to adopt the multikernel [15] philosophy of running a VM per core, and the single-threaded OCaml runtime has fast sequential performance that is ideal for this need. Each Mirage unikernel runs over Xen using a single virtual CPU, and multicore is supported via multiple communicating unikernels over a single instance of Xen.

We did explore applying unikernel techniques in the traditional systems language, C, linking application code with Xen MiniOS, a cut-down `libc`, OpenBSD versions of `libm` and `printf`, and the lwIP user-space network stack. However, we found that a DNS appliance constructed in this way from the high performance NSD DNS server performed considerably worse than the Mirage DNS server, even after several rounds of optimisation (Figure 10). It seems likely that producing even a similarly performing prototype in C would still require very significant engineering effort and would not achieve any of the type-safety benefits.

3.2 PVBoot Library

PVBoot provides start-of-day support to initialise a VM with one virtual CPU and Xen event channels, and jump to an entry function. Unlike a conventional OS, multiple processes and preemptive threading are not supported, and instead a single 64-bit address space is laid out for the language runtime to use. PVBoot provides two memory page allocators, one slab and one extent. The slab allocator is used to support the C code in the runtime; as most code is in OCaml it is not heavily used. The extent allocator reserves a contiguous area of virtual memory which it manipulates in 2MB chunks, permitting the mapping of x86_64 superpages. Memory regions are statically assigned roles, e.g., the garbage collected heap or I/O data pages. PVBoot also has a `domainpoll` function that blocks the VM on a set of event channels and a timeout.

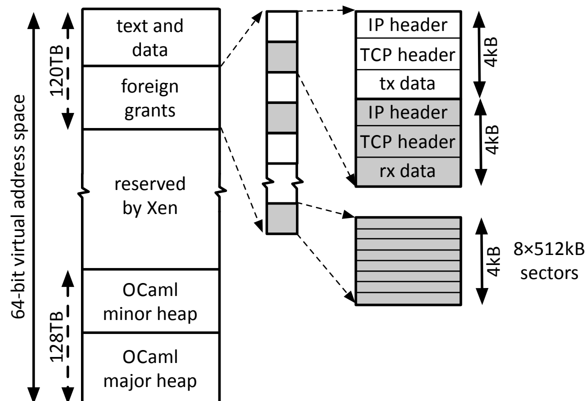


Figure 2: Specialised virtual memory layout of a 64-bit Mirage unikernel running on Xen.

PVBoot provides minimal support for an asynchronous, event-driven VM that sleeps until I/O is available or times out. Device drivers are *all* provided outside PVBoot in type-safe OCaml (§3.4).

3.3 Language Runtime

Mirage executes OCaml code over a specialised language runtime modified in two key areas: memory management and concurrency. Figure 2 shows the memory layout of a Mirage unikernel, divided into three regions: text and data; external I/O pages; and the OCaml heaps. The text and data segments contain the OCaml runtime, linked against PVBoot. This is the only address space available in the kernel. The application's main thread is launched immediately after boot and the VM shuts down when it returns.

The OCaml garbage collector splits the heap into two regions: a fast minor heap for short-lived values, and a large major heap to which longer-lived values are promoted on each minor heap collection. These areas are allocated below the low virtual address space reserved by Xen: the minor heap has a single 2 MB extent that grows in 4 kB chunks, and the major heap has the remainder of virtual memory, growing in 2 MB superpage chunks using the extent allocator. Memory mapping large contiguous areas is usually discouraged to allow Address Space Randomization (ASR) to protect against buffer overflows [16], and so a normal userspace garbage collector maintains a page table to track allocated heap chunks. Mirage unikernels avoid ASR at runtime in favour of a more specialised security model (§2.3), and guarantee a contiguous virtual address space, simplifying runtime memory management.

VMs communicate directly by having the local VM grant memory page access rights to the remote VM via the hypervisor [17]. PVBoot allocates external memory pages from a reserved area of virtual memory, and allocates a small proxy value in the small, fast OCaml minor heap. Mirage provides a library to reference that data from OCaml without requiring a data copy (§3.4). Specialising memory layout to distinguish I/O pages in this way significantly reduces the amount of data that the garbage collector has to scan. This reduced garbage collector pressure is one of two key factors that let the Mirage network stack exhibit predictable performance; the other is pervasive library support for zero-copy I/O (§3.4.1).

To provide concurrency beyond PVBoot's simple domainpoll function, Mirage integrates the Lwt cooperative threading library [18]. This internally evaluates blocking functions into event descriptors to provide straight-line control flow for the developer. Written in pure OCaml, Lwt threads are heap-allocated values, with only the thread main loop requiring a C binding to poll for external events. Mirage provides an evaluator that uses domainpoll to listen for events and wake up lightweight threads. The VM is thus either executing OCaml code or blocked, with no internal preemption or asynchronous interrupts. The main thread repeatedly executes until it completes or throws an exception, and the domain subsequently shuts down with the VM exit code matching the thread return value.

A useful consequence is that most scheduling and thread logic is contained in an application library, and can thus be modified by the developer as they see fit. For example, threads can be tagged with local keys for debugging, statistics or prioritisation, depending on application needs. Thread scheduling is platform-independent with timers stored in a heap-allocated OCaml priority queue, and can be overridden by the application (e.g. we have previously demonstrated the benefit of custom scheduling for the Sqlite library database in an earlier prototype [19]). Only the run-loop is Xen-specific, to interface with PVBoot.

```
cstruct ring_hdr {              || auto-generates these functions:
  uint32_t req_prod;            || set_req_prod : buf → uint32 → unit
  uint32_t req_event;           || get_req_prod : buf → uint32
  uint32_t rsp_prod;            || ...
  uint32_t rsp_event;           || set_stuff : buf → uint64 → unit
  uint64_t stuff                || get_stuff : buf → uint64
} as little_endian              ||
```

Figure 3: Syntax extension mapping C structs *(left)* to OCaml values by autogenerating efficient accessor functions *(right).*

3.4 Device Drivers

Mirage drivers interface to the device abstraction provided by Xen. Xen devices consist of a *frontend* driver in the guest VM, and a *backend* driver that multiplexes frontend requests, typically to a real physical device [20]. These are connected by an event channel to signal the other side, and a single memory page divided into fixed-size request slots tracked by producer/consumer pointers. Responses are written into the same slots as the requests, with the frontend implementing flow control to avoid overflowing the ring. Xen device classes using this model include Ethernet, block storage, virtual framebuffer, USB and PCI.

Manipulating these shared memory rings is the base abstraction for all I/O throughout Mirage. The shared page is mapped into OCaml using the built-in Bigarray module, which wraps externally allocated memory safely into the OCaml heap and makes it available as an array. Reading and writing into the shared page must precisely match the C semantics, and is a relatively slow operation in OCaml since fixed-size integers are boxed values that are heap-allocated before being copied into the shared page.[2] We used `camlp4` to add a new cstruct keyword to OCaml to directly map C structures (Figure 3). Declaring a cstruct generates accessor functions for directly manipulating the external memory array. The extension also handles endian conversion, and is used extensively through the network stack for header parsing [22].

This approach permits Mirage drivers to be implemented as pure OCaml libraries relying on just two extra intrinsics: inline assembly providing read and write memory barriers. The Ring module implements the base protocol and adds an asynchronous threading interface to push requests and wait for responses. Higher-level modules such as Netif and Blkif implement networking and block drivers, and interoperate with unmodified Xen hosts. A beneficial side-effect of our re-implementation of these protocols was to fuzz-test the existing code, and we discovered and reported several edge-case bugs in Linux/Xen as a result, including an important security issue (XSA-39).

[2] The FoxNet network stack also reported this issue in SML [21].

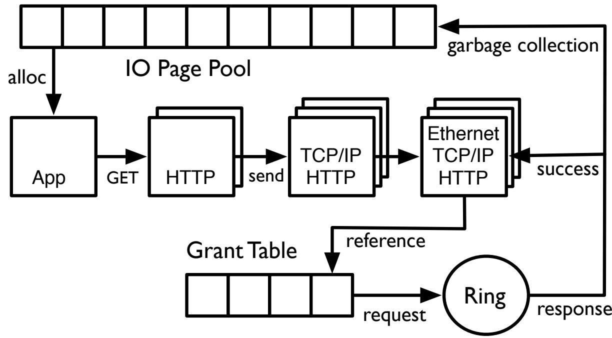


Figure 4: Example zero-copy write for an `HTTP GET`. The application writes its request into an I/O page, and the network stack segments it into fragments to write to the device ring. When a response arrives the pages are collected and the write thread notified.

3.4.1 Zero-Copy Device I/O

The Xen device protocol does not write data directly into the shared memory page, but rather uses it to coordinate passing 4 kB memory pages by reference. Two communicating VMs share a *grant table* that maps pages to an integer offset (called a *grant*) in this table, with updates checked and enforced by the hypervisor. The grant is exchanged over the device driver shared ring, and looked up by the remote VM to map or copy that page into its own address space. Once within the remote (non-Mirage) VM, the data must be transferred into the application. As POSIX APIs do not support zero-copy sockets, this usually entails a second copy from the receiving VM's kernel into the appropriate userspace process.

Mirage unikernels do not have a userspace, so received pages are passed directly to the application without requiring copying. The cstruct library avoids incurring copying overhead by slicing the underlying array into smaller views; once views are all garbage-collected, the array is returned to the free page pool. The network stack can thus safely re-use fragments of incoming network traffic and proxy it directly into another device (e.g., an HTTP proxy) while avoiding having to manage the page manually.

However, there are still a few resources that must be manually tracked and freed, e.g., the contents of the shared grant table between VMs. Mirage uses the OCaml type system to enforce invariants that ensure resources are correctly freed, via higher-order functions that wrap any use of a given resource such as a grant reference. When the function terminates, whether normally via timeout or an unknown exception, the grant reference is freed. As well as preventing resource leaks, particularly on error paths, this allows the scheduler to free resources by cancelling light-weight threads. These composable higher-order functions, also known as *combinators*, are used throughout Mirage to safely expose system resources. Note that these combinators do not entirely eliminate resource leaks, since references that are held in data structures and never removed will remain forever, but the OCaml programming style encourages the use of many small data structures and reusable utility libraries that help prevent such problems.

3.5 Type-Safe Protocol I/O

Mirage implements protocol libraries in OCaml to ensure that *all* external I/O handling is type-safe, making unikernels robust against memory overflows. Protocols are structured as non-blocking parsing libraries with separate client/server logic that spawns lightweight threads. Libraries are accessed via preemptive state handles, enabling multiple protocol stacks within the same unikernel. Table 1 lists the protocols currently implemented in Mirage, sufficient to self-host our website[3] infrastructure, including wiki, blog and DNS servers on the Amazon EC2 public cloud.

Subsystem	*Implemented Protocols*
Core	Lwt, Cstruct, Regexp, UTF8, Cryptokit
Network	Ethernet, ARP, DHCP, IPv4, ICMP, UDP, TCP, OpenFlow
Storage	Simple key-value, FAT-32, Append B-Tree, Memcache
Application	DNS, SSH, HTTP, XMPP, SMTP
Formats	JSON, XML, CSS, S-Expressions

Table 1: System facilities provided as Mirage libraries.

Data arrives to both the network and storage stacks as a stream of discrete packets. Mirage bridges the gap between packets and streams by using channel iteratees [23] that map functions over infinite streams of packets to produce a typed stream. Iterators eliminate many of the fixed-size buffers that are often used in a less tightly coupled conventional kernel/userspace. Chained iterators route traffic directly to the relevant application thread, blocking on intermediate system events if necessary. We now describe the specifics of networking (§3.5.1) and storage (§3.5.2).

3.5.1 Network Processing

The Mirage network stack emphasises application-level control over strict modularity, exposing most details of the underlying protocol to application control. It provides two communication methods: a fast on-host inter-VM vchan transport, and an Ethernet transport for external communication. vchan is a fast shared memory interconnect through which data is tracked via producer/consumer pointers. It allocates multiple contiguous pages for the ring to ensure it has a reasonable buffer and once connected, communicating VMs can exchange data directly via shared memory without further intervention from the hypervisor other than interrupt notifications.[4] vchan is present in upstream Linux 3.3.0 onwards, enabling easy interaction between Mirage unikernels and Linux VMs.

Internet connectivity is more complex: an application links to a protocol library such as HTTP, which links with a TCP/IP network stack, which in turns links to the network device driver. The application reads and writes I/O pages with the protocol library so they can be transmitted directly. When reading packets, the network stack splits out headers and data using cstruct subviews (§3.4.1). Writing data requires more processing as the stack must prepend variable-length protocol headers for TCP, IP and Ethernet before the data payload, and sometimes segment large payloads into smaller fragments for transmission. This is solved via scatter-gather I/O: the network stack allocates a header page for every write, and the network libraries rearrange incoming payloads into sub-views as needed, before writing all the fragments into the device ring as one packet. Figure 4 illustrates this write path.

3.5.2 Storage

Traditional OS kernels layer filesystems over block devices accessed via POSIX sockets or mmap, and coalesce writes into a kernel buffer cache [24]. Applications needing precise control over when a write is actually persisted must either invoke the fsync syscall or explicitly request non-buffered direct access and issue sector-aligned requests. Modern Linux kernels provide libaio for asynchronous block requests only, forcing applications to use different APIs for networking and storage.

In contrast, Mirage block devices share the same Ring abstraction as network devices, using the same I/O pages to provide effi-

[3] `http://openmirage.org/`

[4] When data is continuously flowing between VMs, each side checks for outstanding data before blocking, reducing the number of hypervisor calls.

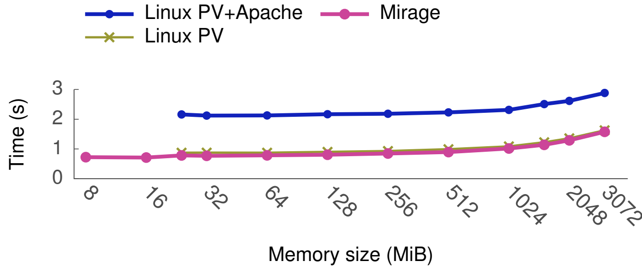


Figure 5: Domain boot time comparison.

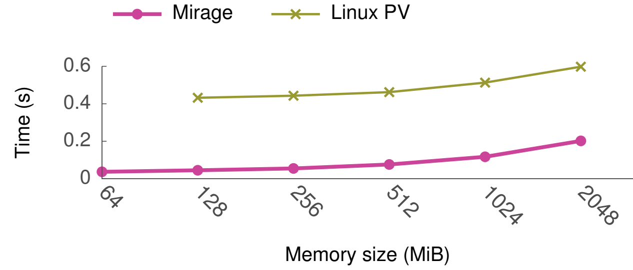


Figure 6: Boot time using an asynchronous Xen toolstack.

cient block-level access, with filesystems and caching provided as OCaml libraries. This gives control to the application over caching policy rather than providing only one default cache policy. Different caching policies can be provided as libraries (OCaml modules) to be linked at build time, with the only built-in policy being that all writes are guaranteed to be direct.

For example, we ported a third-party copy-on-write binary tree storage library[5] to Mirage. This can be used as a storage backend by applications, with caching policy and buffer management being explicitly managed within in the library. Our FAT-32 storage library also implements its own buffer management policy where data reads are returned as iterators supplying one sector at a time. This avoids building large lists in the heap while permitting internal buffering within the library by having it request larger sector extents from the block driver. Finally, we found that our DNS server gained a dramatic speed increase by applying a memoization library to *network* responses (§4); this technique could also be used to implement persistent self-paging of very large datasets [25].

4. Evaluation

As Mirage is a clean-slate implementation of many OS components, we evaluate it against more conventional deployments in stages. We first examine micro-benchmarks (§4.1) to establish baseline performance of key components, followed by more realistic appliances: a DNS server, showing performance of our safe network stack (§4.2); an OpenFlow controller appliance (§4.3); and an integrated web server and database, combining storage and networking (§4.4). Finally, we examine the differences in active LoC and binaries in these appliances, and the impact of dead-code elimination (§4.5).

4.1 Microbenchmarks

These microbenchmarks demonstrate the *potential* benefits of unikernel specialisation by examining performance in simple, controlled scenarios. Evaluations are composed of identical OCaml code executing in different hosting environments: *linux-native*, a Linux kernel running directly on the bare metal with an ELF binary version of the application; *linux-pv*, a Linux kernel running as a paravirtualised Xen domU with an ELF binary version of the application; and *xen-direct*, the application built as a type-safe unikernel, running directly over Xen. We verified that CPU-bound applications are unaffected by unikernel compilation as expected, as the hypervisor architecture only affects memory and I/O.

4.1.1 Boot Time

Unikernels are compact enough to boot and respond to network traffic in real-time.

Mirage generates compact VMs which boot very quickly. Figure 5 compares boot times for a Mirage webserver against a minimal Linux kernel, and a more realistic *linux-pv* Debian Linux running Apache2. Time is measured from startup to the point where boot is complete, signalled by the VM sending a special UDP packet to the control domain. The minimal Linux kernel measures this "time-to-userspace" via an `initrd` that calls the `ifconfig` ioctls directly to bring up a network interface before explicitly transmitting a single UDP packet. The more realistic Debian Linux running Apache2 uses the standard Debian boot scripts and measures time-to-userspace by waiting until Apache2 startup returns before transmitting the single UDP packet. The Mirage unikernel transmits the UDP packet as soon as the network interface is ready. As the memory size increases, the proportion of Mirage boot time due to building the domain also increases to approximately 60% for memory size 3072 MiB. Mirage matches the minimal Linux kernel, booting in slightly under half the time of the Debian Linux.

It is difficult to distinguish between the minimal Linux VM and Mirage in Figure 5. The boot time is skewed by the Xen control stack synchronously building domains, since latency isn't normally a prime concern for VM construction. We modified the Xen toolstack to support parallel domain construction, and isolate the VM startup time in Figure 6. This clearly distinguishes the differences between the Mirage unikernel and Linux: Mirage boots in *under 50 milliseconds*. Such fast reboot times mitigate the concern that redeployment by reconfiguration is too heavyweight, as well as opening up the possibility of regular micro-reboots [14].

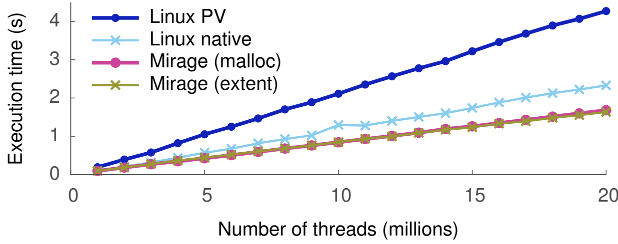
4.1.2 Threading

Garbage collected heap management is more efficient in a single address-space environment. Thread latency can be reduced by eliminating multiple levels of scheduling.

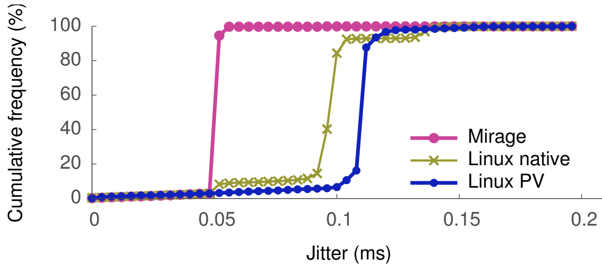
Figure 7a benchmarks thread construction time, showing the time to construct millions of threads in parallel where each thread sleeps for between 0.5–1.5 seconds and then terminates. The *linux-pv* userspace target, which most closely mirrors a conventional cloud application, is slowest with the same binary running on native Linux coming in next. The two *xen-* targets using the different memory allocators perform notably better due to the test being limited by the GC speed: thread construction occurs on the OCaml heap so creating millions of threads triggers regular compaction and scanning. The *xen-* runtime is faster due to the specialised address space layout described earlier (§2). There is little extra benefit to using superpages (*xen-extent* cf. *xen-malloc*), as the heap grows once to its maximum size and never subsequently shrinks.

We also evaluated the precision of thread timers: a thread records the domain wallclock time, sleeps for 1–4 seconds and records the difference between the wallclock time and its expected wakeup time. Figure 7b plots the CDF of the jitter, and shows that the unikernel target provides both lower and more predictable latency when waking up millions of parallel threads. This is due simply to the lack of userspace/kernel boundary eliding Linux's syscall overhead.

[5] `https://github.com/Incubaid/baardskeerder`



(a) Creation times.



(b) Jitter for 10^6 parallel threads sleeping and waking after a fixed period.

Figure 7: Mirage thread performance.

4.1.3 Networking and Storage

Unikernel low-level networking performs competitively to conventional OSs, despite being fully type-safe. Library-based block management performs on par with a conventional layered storage stack.

As a simple latency test against the Linux stack we flooded 10^6 pings from the Linux *ping* client running in its own VM to two targets: a standard Linux VM, and a Mirage unikernel with the Ethernet, ARP, IPv4 and ICMP libraries. This stress tests pure header parsing without introducing any userspace element for Linux. As expected, Mirage suffered a small (4–10%) increase in latency compared to Linux due to the slight overhead of type-safety, but both survived a 72-hour flood ping test.

We compared the performance of Mirage's TCPv4 stack, implementing the full connection lifecycle, fast retransmit and recovery, New Reno congestion control, and window scaling, against the Linux 3.7 TCPv4 stack using `iperf`. All hardware offload was disabled to provide the most stringent test of Mirage: the naturally higher overheads of implementing low-level operations in OCaml rather than C mean that hardware offload (particularly TCP segmentation) disproportionately improves Mirage's performance compared to Linux. Performance is on par with Linux: Mirage's receive throughput is slightly higher due to the lack of a userspace copy, while its transmit performance is lower due to higher CPU usage. Both Linux and Mirage can saturate a gigabit network connection, and we expect that adding transmit hardware offload support will allow even our experimental TCP stack to 10 Gb/s performance.

Configuration	*Throughput [std. dev.] (Mbps)*			
	1 flow		10 flows	
Linux to Linux	1590	[9.1]	1534	[74.2]
Linux to Mirage	1742	[18.2]	1710	[15.1]
Mirage to Linux	975	[7.0]	952	[16.8]

Figure 8: Comparative TCP throughput performance with all hardware offload disabled.

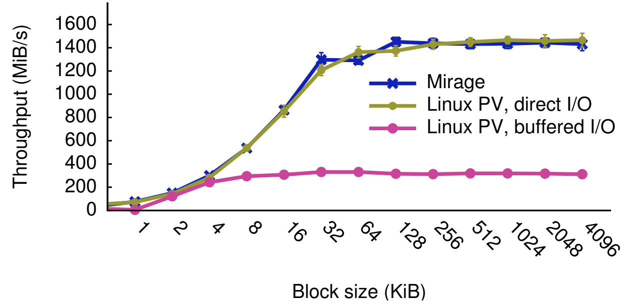


Figure 9: Random block read throughput, +/- 1 std. dev.

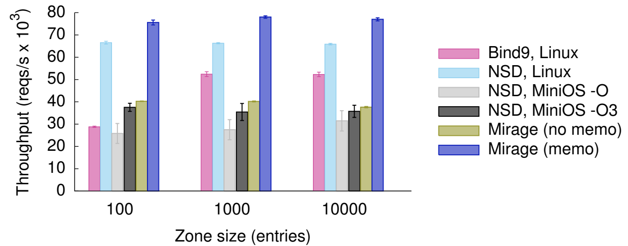


Figure 10: DNS performance with increasing zone size.

Figure 9 shows a simple random read throughput test using *fio* of a fast PCI-express SSD storage device, comparing a Mirage *xen-direct* appliance against a Linux RHEL5 kernel (2.6.18) using buffered and direct I/O. Again, as expected, the Linux direct I/O and Mirage lines are effectively the same: both use direct I/O and so impose very little overhead on the raw hardware performance. However, the performance impact of the Linux buffer cache *is* notable: it causes performance to plateau at around 300 MB/s in contrast to the 1.6 GB/s achieved if the buffer cache is avoided. The lack of a buffer cache is not significant for the appliances we target: such applications already manage their own storage.

4.2 DNS Server Appliance

The Mirage DNS Server appliance contains the core libraries, the Ethernet, ARP, IP, DHCP and UDP libraries from the network stack, and a simple in-memory filesystem storing the zone in standard Bind9 format. Figure 10 compares the throughput of the Mirage appliance on Xen against two best-of-breed nameservers: Bind 9.9.0, a mature and widely used DNS server; and NSD 3.2.10, a recent high performance implementation.

Bind achieves 55 kqueries/s for reasonable zone file sizes.[6] As a more recent rewrite focused on performance, NSD does better, achieving around 70 kqueries/s. The Mirage appliance initially performed very poorly, but dramatically improved when we introduced memoization of responses to avoid repeated computation. A simple 20 line patch, this increased performance from around 40 kqueries/s to 75–80 kqueries/s. Errorbars indicate unbiased estimates of $\mu \pm \sigma$ across 10 runs.

There is other evidence [26] that algorithmic performance improvements substantially exceed those due only to machine-level optimization, and Mirage's use of functional programming provides an effective platform for further experimentation in this regard. A further example is DNS label compression, notoriously

[6] We were unable to determine the cause of Bind's poor performance with small zone sizes, but the results are consistently reproducible.

tricky to get right as previously seen label fragments must be carefully tracked. Our initial implementation used a naive mutable hashtable, which we then replaced with a functional map using a customised ordering function that first tests the size of the labels before comparing their contents. This gave around a 20% speedup, as well as securing against the denial-of-service attack where clients deliberately cause hash collisions.

DNS also provides a good example where Mirage type-safety should bring significant security benefits. For example, in the last 10 years the Internet Systems Consortium has reported 40 vulnerabilities in the Bind software.[7] Of these, 25% were due to memory management errors, 15% to poor handling of exceptional data states, and 10% to faulty packet parsing code, *all* of which would be mitigated by Mirage's type-safety.

Finally, the other main benefit of Mirage is shown by comparing the size of the Linux and Mirage appliances: the Mirage appliance is 183.5 kB in size compared with 462 MB in-use for the Linux VM image. While some of this difference can be accounted to the fact that our libraries do not implement the complete feature-set of BIND9 or NSD, we do include all features required by the *queryperf* test suite and the Mirage appliance is sufficient to self-host the project infrastructure online.

We also tested both NSD and BIND compiled in libOS mode with the *newlib-1.16* embedded libc, the lwIP-1.3.0 network stack and the standard Xen-4.1 MiniOS device drivers. Performance was significantly lower than expected with further benchmarking revealing that this is due to unexpected interactions between MiniOS `select(2)` scheduling and the `netfront` driver. Our experiences with the C libOS libraries reinforced our notion that such programming is rather fragile – using embedded systems libraries often means giving up performance (e.g., optimised `libc` assembly is replaced by common calls) – and a more fruitful approach would be to break the Linux kernel into a libOS as Drawbridge does for Windows 7 [3].

4.3 OpenFlow Controller Appliance

OpenFlow is a software-defined networking standard for Ethernet switches [27]. It defines an architecture and a protocol by which the *controller* can manipulate *flow tables* in Ethernet switches, termed *datapaths*. Mirage provides libraries implementing an OpenFlow protocol parser, controller, and switch. By linking against the controller library, appliances can exercise direct control over hardware and software OpenFlow switches, subject to any network administration policies in place. Conversely, by linking against the switch library, an appliance can be controlled as if it were an OpenFlow switch, useful in scenarios where the appliance provides network layer functionality, e.g., acts as a router, switch, firewall, proxy or other middlebox. As software implementations, these libraries can be extended according to specific appliance needs, e.g., to enable flows to be identified by the DNS domain of either endpoint, or defined in terms of HTTP URLs.

We benchmark our OpenFlow implementation using the OFlops platform [28]. For the controller benchmark we use *cbench* to emulate 16 switches concurrently connected to the controller, each serving 100 distinct MAC addresses. Experiments run on a 16-core AMD server with 40 GB RAM, and each controller is configured to use a single thread. We measure throughput in requests processed per second in response to a stream of *packet-in* messages produced by each emulated switch under two scenarios: *batch*, where each switch maintains a full 64 kB buffer of outgoing *packet-in* messages; and *single*, where only one *packet-in* message is in flight from each switch. The first measures the absolute throughput when servicing requests, and the second measures throughput of the controller when serving connected switches fairly.

[7] `http://www.isc.org/advisories/bind`

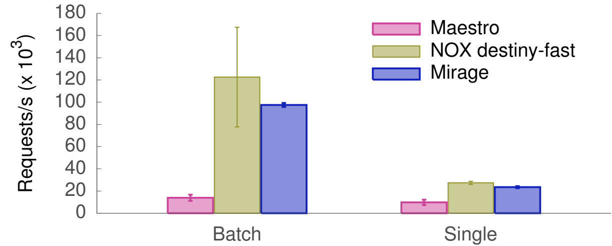


Figure 11: OpenFlow controller performance ($\mu \pm \sigma$).

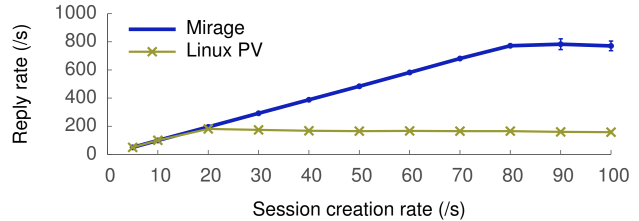


Figure 12: Simple dynamic web appliance performance.

Figure 11 compares the *xen-direct* Mirage controller against two existing OpenFlow controllers: Maestro [29], an optimised Java-based controller; and the optimised C++ *destiny-fast* branch of NOX [30], one of the earliest and most mature publicly available OpenFlow controllers. Unsurprisingly, the optimised NOX branch has the highest performance in both experiments, although it does exhibit extreme short-term unfairness in the batch test. Maestro is fairer but suffers significantly reduced performance, particularly on the "single" test, presumably due to JVM overheads. Performance of the Mirage appliance falls between NOX and Maestro, showing that Mirage manages to achieve most of the performance benefits of optimised C++ which retaining the high-level language features such as type-safety.

4.4 Dynamic Web Server Appliance

Our final appliance implements a simple "Twitter-like" service. It maintains an in-memory database of tweets and is exercised through two API calls: one to `GET` the last 100 tweets for an individual, and the other to `POST` a tweet to an individual. The Mirage implementation integrates the third-party Baardskeerder B-tree storage library, ported to Mirage with only a small patch to use the `Block` APIs instead of UNIX I/O. We compare the Mirage unikernel implementation against a Linux-based appliance running *nginx*, *fastCGI* and *web.py*. We used the *httperf* benchmark tool as a client on the same physical box with separate dedicated cores, connecting over the local bridge to avoid the possibility of the Ethernet becoming the bottleneck. Each *httperf session* issues 10 requests: 1 tweet and 9 'get last 100 tweets'.

Figure 12 shows the results. The unikernel implementation clearly scales much better: scaling is linear up to around 80 sessions (800 requests – each session consists of 10 HTTP requests, 9 GETs and 1 POST) before it becomes CPU bound. In contrast, Linux VM performance is much worse, reaching its limit at around 20 sessions. For comparison, the same Linux VM serving two clients just a single static page via *nginx* achieves up to 5,000 requests/s before hitting CPU and *fd* limits. Although substantially higher than the (unoptimised) Mirage implementation, there are other benefits to Mirage discussed earlier: smaller memory footprint (32 MB against 256 MB), type-safety and security.

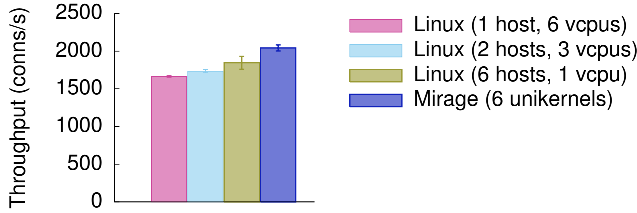


Figure 13: Static page serving performance, comparing Mirage and Apache2 running on Linux.

Figure 13 compares performance of the Mirage web appliance against a standard Linux VM running Apache2 using the mpm-worker backend with the number of workers sized to the number of vCPUs, serving a single static page. The Linux VM was run in three configurations: a single VM given 6 vCPUs, two VMs each given 3 vCPUs and finally 6 VMs each given a single vCPU. As Mirage unikernels do not support multi-core, only one configuration was run of 6 unikernels each with a single vCPU. The results show first, that scaling out appears to improve the Apache2 appliance performance more than having multiple cores and second, that the Mirage unikernels exceed the Apache2 appliance in all cases.

4.5 Code and Binary Size

Direct comparison of lines-of-code (LoC) is rarely meaningful due to factors including widespread use of conditional compilation and complex build systems. We attempt to control for these effects by configuring according to reasonable defaults, and then pre-processing to remove unused macros, comments and whitespace. In addition, to attempt a fair comparison against the 7 million LoC left in the Linux tree after pre-processing, we ignore kernel code associated with components for which we have no analogue, e.g., the many architectures, network protocols, and filesystems that Linux supports. As we are concerned with network-facing guest VMs that share the underlying hypervisor, we do not include LoC for Xen and its management domains; these can be separately disaggregated as desired [14, 31].

Figure 14a shows LoC for several popular server components, computed by the *cloc* utility. Even after removing irrelevant code, a Linux appliance involves at least 4–5x more LoC than a Mirage appliance. Note that, although the Mirage libraries are not yet as feature-rich as the industry-standard C applications, their library structure ensures that unused dependencies can be shed at compile-time even as features continue to be added. For example, if no filesystem is used, then the entire set of block drivers are automatically elided. Performing such dependency analysis across the kernel and userspace is non-trivial in a Linux distribution.

Compiled binary size is another effective illustration of this, and Table 2 gives binary sizes for the benchmark appliances. The first column used the default OCaml dead-code elimination which drops unused modules, and the second uses ocamlclean,[8] a more extensive custom tool which performs dataflow analysis to drop unused functions *within* a module if not otherwise referenced; this is safe due to the lack of dynamic linking in Mirage [32]. Either way, all Mirage kernels are significantly more compact than even a cut-down embedded Linux distribution, without requiring any work on the part of the programmer beyond using the Mirage APIs to build their application.

[8] http://github.com/avsm/ocamlclean

Appliance	*Binary size (MB)*	
	Standard build	Dead code elimination
DNS	0.449	0.184
Web Server	0.673	0.172
OpenFlow switch	0.393	0.164
OpenFlow controller	0.392	0.168

Table 2: Sizes of Mirage unikernels, before and after dead-code elimination. Configuration and data are compiled directly into the unikernel.

5. Discussion & Related Work

We next discuss the relationship of both unikernels and Mirage to the previous work on which they build, from fields such as type-safety, library OSs and security.

5.1 Type-safe and library OSs

The desire to apply type-safety in the OS is not new: type-safe OSs have been built in a range of languages including Haskell [33, 34], Java [35], Standard ML [21], C#/.Net [4], and Modula-3 [36]. The last of these, SPIN [36], is perhaps the closest in nature to Mirage: SPIN is an extensible OS that relies on Modula-3 type-safety to dynamically link extensions into the kernel. More recently, Singularity [4] restructured the OS to sit above the Common Language Runtime, achieving many similar type-safety benefits. The unikernel approach is somewhat orthogonal: it proposes a restructuring of the OS to specifically target services hosted on the public cloud, which benefits from but does not mandate type-safety. The particular implementation presented here, Mirage, also uses an extremely portable functional language which means the Mirage core and network stack can even be compiled to JavaScript for execution on *node.js*;[9] and ports to ARM and a FreeBSD kernel module are functional in an alpha state.

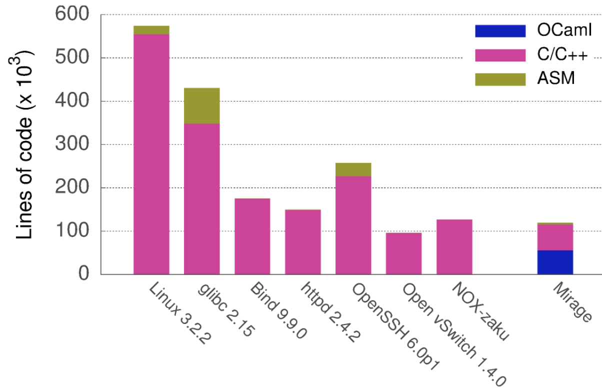
Restructuring the OS into a set of libraries that are linked into the application has been explored extensively. Exokernel [1], Nemesis [2], and Scout [37] all pursued this approach with considerable success. Indeed, the specific technique of page re-use for efficient network I/O (§3.5.1) is directly inspired by the high-performance Cheetah web server for the Exokernel [38]. More recently, Drawbridge [3] adapts Windows to be a library operating system where applications communicate via networking protocols, showing that this approach scales to real commercial operating systems. However, previous library OSs suffered from poor hardware support with device drivers either needing to be written from scratch or wrapped in a compatibility layer [39]. Unikernels use similar libOS techniques but targets the cloud as a deployment platform, with our Mirage implementation using the Xen hypervisor as a low-level common hardware interface.

Finally, Libra [5] adapts the JVM to sit directly above Xen, targeting Java workloads specifically. In particular, Libra uses a gateway server running in a standard Linux VM to provide access to standard OS services such as a networking stack and filesystem IO. Application-hosting VMs access these services by communicating with the other VM via the gateway process. In contrast, Unikernels are more highly-specialised but also more complete, providing single-purpose appliance VMs that directly support standard network and filesystem protocols.

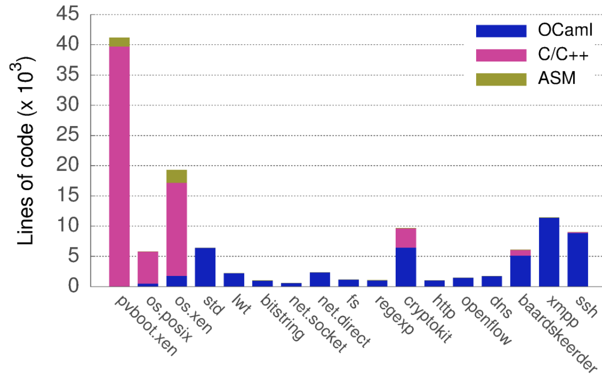
5.2 Legacy Support

Perhaps the most obvious potential problem with the unikernel approach is how best to support legacy systems. The extreme position we take does appear to require a great deal of re-implementation,

[9] Admittedly without a specific purpose in mind, though it is useful for educational purposes and ensuring portability.



(a) Key cloud components *vs*. the Mirage unikernel codebase.



(b) Key components of the Mirage unikernel.

Figure 14: Comparison of lines-of-active-code counts.

possibly in an unfamiliar language if using our Mirage prototype. Consider for example the standard implementation of the SSL protocol in the OpenSSL [40] library, or storage formats such as *ext2* that are only properly documented in their standard implementation's code: a complete rewrite of such key components cannot be lightly undertaken! On the other hand, these security-critical libraries currently present complex APIs that often lead to serious vulnerabilities [41], and higher-level functional interfaces would be very valuable alternatives to existing C implementations.

Alternative approaches retrofit type-safety to existing codebases, using, e.g., tools like CIL or CCured [42, 43]. Unfortunately, it turns out to be considerable work to implement specialisation techniques for the particular underlying platforms, and it would be hard to integrate the results of these into a unikernel. At the other end of the spectrum, recent tools such as HipHop [44] take steps toward the unikernel approach, translating PHP code into C++ and then compiling a single binary containing the entire PHP application. One can envisage further specialising such binaries into unikernels, though the benefits of static type-safety would not accrue with PHP.

We take an approach pioneered by the Flux OSKit [39], which encapsulated existing code to port it into the new system. Flux did so by targeting the multiboot bootloader standard and then wrapping device drivers from systems such as Linux to fit within it, and encapsulation is a very promising technique to apply to larger cloud components. For example, 'big data' processing systems, such as Map-Reduce, Hadoop, and Dryad [45–47] are typically structured as a set of intercommunicating processes, farmed out within a datacenter. Each of these processes could be encapsulated as a single VM and message-passing between VMs implemented via Vchan. This approach is similar to UNIX privilege separation [48], and provides an incremental deployment path, ensuring existing reliably engineered components can continue to be used and that multiple, new, untested components need not be introduced all at once.

5.3 Deeper, Higher, Broader

One shortcoming of the Mirage unikernel implementation is that it still relies on a great deal of unsafe code, notably the Xen hypervisor and management domain, *dom0*. This raises the question, can we push the unikernel further down? This would require either re-implementing the hypervisor itself or, for more resource constrained platforms, forgoing it completely, focusing instead on a single encapsulated application. In both cases, much of the current approach – such as the I/O stack – would carry over, but native support for many-core hardware would require additional work. One

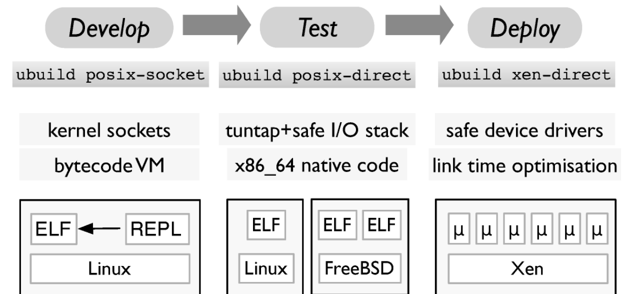


Figure 15: Specialising a Mirage application through recompilation alone, from interactive UNIX Read-Eval-Print Loop, to reduced dependency on the host kernel, and finally a unikernel VM.

interesting possibility is to build on the Barrelfish [15] software. Existing cloud orchestration layers such as OpenStack or Eucalyptus [49] exhibit high latency when manipulating small VMs, when compared to processes running with a single OS. Unikernels depend on running multiple VMs for parallelization, and so improvements will be needed in this space. Finally, the use of cooperative threading does require some form of broader management system, as a single bug can completely deadlock an entire appliance.

In tandem, the question arises as to how high the level of abstraction can be raised. Internet protocols are relatively straightforward to implement due to the comprehensive and reasonably accurate RFC specifications. However higher level services (e.g., filesystems, cryptographic libraries and middleware) are often only informally specified at best. Fortunately, tools to specify correct implementations are being developed by the community [50, 51], and some theorem provers can extract OCaml code from these specifications [52]. Work exploring application of verification techniques to key system components, notably the compiler backend and an OS microkernel [53, 54] is also promising.

Finally, thinking more broadly, development for and deployment to the cloud raise questions concerning workflow and orchestration. Real-world cloud computing deployments already have increasingly sophisticated orchestration systems to manage the remote deployment of VMs, and we have designed Mirage to take advantage of this (e.g. with small deployment binary sizes). Figure 15 illustrates how Mirage code is locally developed and tested on a UNIX-like system before compilation into a unikernel for deployment to the cloud. The developer first builds the posix-socket

target, which links in the bytecode interpreter, makes use of the host kernel's networking stack, and maps keys in the k/v store to local files. This provides a familiar development environment in which to debug and profile application logic. Next, the developer applies the first specialisation step, building for the `posix-direct` target. This removes dependency on the standard OS sockets implementation, instead linking in the unikernel network stack (§3.5), and generating a native-code binary that uses `tuntap` to handle Ethernet traffic, and compile configuration files directly into the binary. The result is that I/O processing is performed within the application, albeit inefficiently due to the data copying induced by `tuntap`. This allows testing of the unikernel locally using UNIX development tools rather than the sparse facilities available when debugging microkernels, facilitating isolation of bugs in application logic, Mirage libraries, or due to interactions between the two.

Finally, the developer further specialises via the `xen-direct` target, generating a standalone unikernel. Application code is cross-compiled and linked with the safe Mirage device drivers, and dead-code elimination: if the application doesn't use a component, e.g., TCP, it will not be compiled in. The result is a VM image bootable on Xen locally or via a public cloud service.[10] The image is much smaller than an equivalent Linux-based distribution (Figure 14), and crucially, *all* I/O traffic is processed by type-safe code, with the performance and security benefits of the specialised runtime.

6. Conclusions

We presented the unikernel approach to significantly improving the safety and efficiency of building and deploying appliances for the cloud. Building on earlier work in library operating systems, we contribute the design and evaluation of a statically type-safe OCaml prototype for a libOS, including a complete clean-slate set of protocol libraries which ensure that deployed unikernels are memory-safe from the ground-up. Our approach also optionally extends the hypervisor with special support for such dedicated VMs to improve runtime security and boot latency.

Through our experimental implementation, we showed how security and efficiency benefits can be achieved by relaxing source-level backwards compatibility requirements by switching to novel programming styles such as those afforded by OCaml. Our evaluation showed that these benefits come at little to no cost to performance in all cases, and can actually improve performance in some. Overall we have demonstrated that the marriage of commodity cloud computing platforms with modern language runtimes is fruitful. Unikernels can serve as a platform for a range of future research exploring how to better take advantage of plentiful cloud computing resources, particularly for distributed applications that benefit from horizontal scaling across cores and hosts. Code for the Mirage prototype and our experiment scripts are open-source, available for download under a BSD-style license from `http://openmirage.org`.

Acknowledgments

We thank Pierre Chambart and Fabrice Le Fessant for contributing OCaml compiler optimizations, and Raphael Proust and Gabor Pali for the Javascript and kFreeBSD targets. Malte Schwarzkopf, Derek Murray, Robert Watson, Jonathan Ludlam, Derek McAuley, Peter G. Neumann, Ewan Mellor, Fernando Ramos, Tim Harris, Peter Sewell, Andrew Moore, Tom Kelly, David Chisnall, Jon Howell, Stephen Kell, Yaron Minsky, Marius A. Eriksen, Tim Deegan, Alex Ho and the anonymous ASPLOS reviewers all contributed valuable feedback. This work was primarily supported by Horizon Digital Economy Research, RCUK grant EP/G065802/1, and a portion was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), 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 the Defense Advanced Research Projects Agency or the Department of Defense.

[10] Mirage currently automates this process for Amazon EC2, wrapping custom kernels in a block device and registering them as AMIs.

References

[1] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, CO, USA, December 3–6 1995.

[2] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[3] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, Newport Beach, CA, USA, March 5–11 2011.

[4] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[5] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: a library operating system for a JVM in a virtualized execution environment. In *Proc. 3rd International Conf. on Virtual Execution Environments (VEE)*, pages 44–54, San Diego, CA, USA, June 13–15 2007. ACM.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, USA, October 19–22 2003.

[7] US-CERT/NIST. CVE-2012-1182, February 2012.

[8] US-CERT/NIST. CVE-2012-2110, April 2012.

[9] Eelco Dolstra, Andres LÖh, and Nicolas Pierron. Nixos: A purely functional Linux distribution. *J. Funct. Program.*, 20(5-6):577–615, November 2010.

[10] Galen Hunt, Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, James Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. *SIGOPS Operating Systems Review*, 41(3):341–354, March 2007.

[11] Theo De Raadt. Exploit mitigation techniques. `http://www.openbsd.org/papers/auug04`, 2004.

[12] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group: perspectives and perceptions. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 87–92, Baltimore, Maryland, USA, September 27–29 2010.

[13] Thomas Gazagnaire and Vincent Hanquez. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. *SIGPLAN Notices*, 44(9):203–214, August 2009.

[14] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–202, Cascais, Portugal, October 23–26 2011.

[15] A. Baumann, P. Barham, P. 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 *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, Big Sky, MT, USA, October 11–14 2009.

[16] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, Washington DC, USA, October 25–29 2004.

[17] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proc. 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Boston, MA, USA, October 7–13 2004.

[18] Jérôme Vouillon. Lwt: a cooperative thread library. In *Proc. 2008 ACM SIGPLAN workshop on ML*, pages 3–12, Victoria, BC, Canada, September 21 2008.

[19] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the LAMP: Software specialisation for the cloud. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*, June 2010.

[20] Andrew Warfield, Keir Fraser, Steven Hand, and Tim Deegan. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference*, pages 379–382, April 10–15 2005.

[21] Edoardo Biagioni. A Structured TCP in Standard ML. In *Proc. ACM SIGCOMM*, pages 36–45, London, UK, Aug. 31–Sep. 02 1994.

[22] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan. Melange: creating a "functional" Internet. *SIGOPS Operating Systems Review*, 41(3):101–114, 2007.

[23] Oleg Kiselyov. Iteratee IO: safe, practical, declarative input processing. `http://okmij.org/ftp/Streams.html`, 2008.

[24] Chuck Silvers. UBC: an efficient unified I/O and memory caching subsystem for NetBSD. In *Proc. USENIX Annual Technical Conference*, pages 285–290, San Diego, CA, USA, June 18–23 2000.

[25] Steven M. Hand. Self-paging in the Nemesis operating system. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, February 22–25 1999.

[26] President's Council of Advisors on Science and Technology. Report to the President and Congress: Designing a Digital Future: Federally Funded R&D in Networking and IT, December 2010.

[27] OpenFlow Consortium. OpenFlow. `http://openflow.org/`.

[28] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Proc. Passive and Active Measurements Conference (PAM)*, Vienna, Austria, March 12–14 2012.

[29] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A system for scalable OpenFlow control. Technical Report TR-10-11, Rice University.

[30] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. *SIGCOMM Computer Communications Review*, 38:105–110, July 2008.

[31] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen security through disaggregation. In *Proc. 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 151–160, Seattle, WA, USA, March 5–7 2008.

[32] B. Vaugon, Philippe Wang, and Emmanuel Chailloux. Les microcontrôleurs pic programmés en Objective Caml. In *Vingt-deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, volume Studia Informatica Universalis, pages 177–207. Hermann, 2011.

[33] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A Principled Approach to Operating System construction in Haskell. *SIGPLAN Notices*, 40(9):116–128, 2005.

[34] Galois Inc. HalVM. `http://halvm.org/`.

[35] Oracle. GuestVM. `http://labs.oracle.com/projects/guestvm/shared/guestvm/guestvm/index.html`.

[36] 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. *SIGOPS Operating Systems Review*, 29(5):267–283, December 1995.

[37] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–167, Seattle, WA, United States, October 28–31 1996.

[38] F. Kaashoek, D. Engler, G. Ganger, H. Brice no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, Saint Malo, France, October 5–8 1997.

[39] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: a substrate for kernel and language research. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–51, Saint Malo, France, October 5–8 1997.

[40] The OpenSSL Project. OpenSSL. `http://openssl.org/`.

[41] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *Proc. 19th ACM Conference on Computer and Communications Security (CCS)*, pages 38–49, Raleigh, NC, USA, October 16–18 2012.

[42] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. 11th International Conference on Compiler Construction (CC)*, LNCS 2304, pages 213–228, Grenoble, France, April 8–12 2002.

[43] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–139, January 16–18 2002.

[44] Facebook. HipHop for PHP. `https://github.com/facebook/hiphop-php/wiki/`, February 2010.

[45] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, pages 137–150, San Francisco, CA, USA, December 6–8 2004.

[46] Apache. Hadoop. `http://hadoop.apache.org`, April 2012.

[47] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21–23 2007.

[48] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proc. 12th USENIX Security Symposium (SSYM)*, pages 231–242, Washington DC, USA, August 4–8 2003.

[49] Bill Childers. Build your own cloud with Eucalyptus. *Linux J.*, 2010(195), July 2010.

[50] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *Proc. ACM Workshop on Formal Methods in Security Engineering (FMSE)*, page 41, Fairfax, Virginia, USA, November 2 2007.

[51] Reynald Affeldt, David Nowak, and Yutaka Oiwa. Formal network packet processing with minimal fuss: invertible syntax descriptions at work. In *Proc. 6th Workshop on Programming Languages meets Program Verification (PLPV)*, pages 27–36, January 24 2012.

[52] Nicolas Oury. Observational equivalence and program extraction in the Coq proof assistant. In *Proc. 6th International Conference on Typed Lambda Calculi and Applications (TLCA)*, LNCS 2701, pages 271–285, Valencia, Spain, June 10–12 2003.

[53] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, Charleston, SC, USA, January 11–13 2006.

[54] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, USA, October 11–14 2009.