Jitsu: Just-In-Time Summoning of Unikernels

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire,
David Sheets, Dave Scott,[1] Richard Mortier, Amir Chaudhry,
Balraj Singh, Jon Ludlam,[1] Jon Crowcroft and Ian Leslie
University of Cambridge, Citrix Systems UK Ltd[1]

Abstract. Network latency is a problem for all cloud services. It can be mitigated by moving computation out of remote datacenters by rapidly instantiating local services near the user. This requires an embedded cloud platform on which to deploy multiple applications securely and quickly. We present *Jitsu*, a new Xen toolstack that satisfies the demands of secure multi-tenant isolation on resource-constrained embedded ARM devices. It does this by using *unikernels*: lightweight, compact, single address space, memory-safe virtual machines (VMs) written in a high-level language. Using fast shared memory channels, Jitsu provides a directory service that launches unikernels in response to network traffic and masks boot latency. Our evaluation shows Jitsu to be a power-efficient and responsive platform for hosting cloud services in the edge network while preserving the strong isolation guarantees of a type-1 hypervisor.

1 Introduction

The benefits of cloud hosting are clear: dynamic resource provisioning, lower capital expenditure, high availability, centralised management. Unfortunately, all services architected and deployed in this way inevitably suffer the same problem: *latency*. Physical separation between remote datacenters where processing occurs and users of these services, imposes unavoidable minimum bounds on network latency. Recent developments in augmented reality (e.g., Google Glass [17]) and voice control (e.g., Apple's Siri) particularly suffer in this regard.

Concurrent with the move of services to the cloud, we are now seeing uptake of the "Internet-of-Things" (IoT), giving rise to our second concern: *integrity*. These devices often rely on the network for their operation but many of the devices we use daily suffer from an unrelenting stream of security exploits, including routers [8], buildings [13] and automobiles [5]. The future success of IoT platforms being deployed in edge networks depends on the convenience of secure multi-tenant isolation that the public cloud utilises.

The widely deployed Xen hypervisor [2] enforces isolation between multiple tenants sharing physical machines. Xen recently added support for hardware virtualized ARM guests, opening up the possibility of building an *embedded cloud*: a system of distributed low-power devices, deployed near users, able to host applications delivering real-time services directly via local networks. There has been a steady increase in ARM boards featuring a favourable energy/price/speed trade-off for constructing embedded systems (e.g., the Cubieboard2 has 1GB RAM, a dual-core A20 ARM CPU and costs £ 39).

We present *Jitsu*, a system for securely managing multi-tenant networked applications on embedded infrastructure. Jitsu re-architects the Xen toolstack to lower the resource overheads of manipulating virtual machines (VMs), overcoming current limitations that prevent Xen from becoming an effective platform for building embedded clouds. Rather than booting conventional VMs, Jitsu services network requests with low latency using unikernels [27] as the unit of deployment. These are small enough to be booted in a few hundred milliseconds, a latency that Jitsu further masks through connection hand-off. The MirageOS unikernels [25] that we use are also secure enough to survive inexpertly managed network-facing deployment.

Jitsu uses the virtual hardware abstraction layer provided by the Xen type-1 hypervisor, adding a new control toolstack that eliminates bottlenecks in VM management (Figure 1). Although developed with unikernels in mind, it preserves sufficient compatibility that many of its benefits apply equally to generic (e.g., Linux or FreeBSD) VMs targeting either ARM or x86.

The specific contributions of this paper are thus: a description of how to build efficient, secure unikernels on the new open-source Xen/ARM (§2); an explanation of the Jitsu Xen toolstack architecture (§3); a comparison of it against other application containment techniques (§4); and finally application deployment scenarios and discussion of the broader lessons learnt (§5).

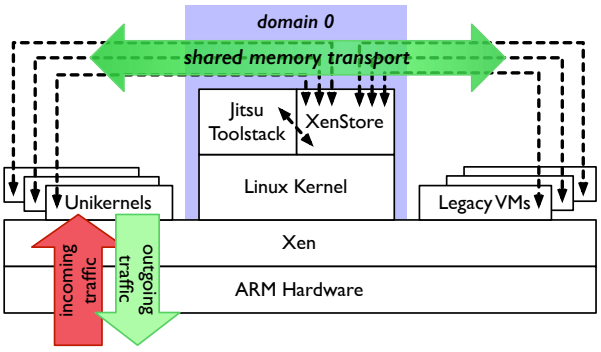


Figure 1: Jitsu architecture: external network connectivity is handled solely by memory-safe unikernels connected to general purpose VMs via shared memory.

2 Embedded Unikernels

Building software for embedded systems is typically more complex than for standard platforms. Embedded systems are often power-constrained, impose soft real-time constraints, and are designed around a monolithic firmware model that forces whole system upgrades rather than upgrade of constituent packages. To date, general-purpose hypervisors have not been able to meet these requirements, though microkernels have made inroads [9].

Several approaches to providing application isolation have received attention recently. As each provides different trade-offs between security and resource usage, we discuss them in turn (§2.1), motivating our choice of unikernels as our unit of deployment. We then outline the new Xen/ARM port that uses the latest ARM v7-A virtualization instructions (§2.2) and provide details of our implementation of a single-address space ARM unikernel using this new ABI (§2.3).

2.1 Application Containment

Strong isolation of multi-tenant applications is a requirement to support the distribution of application and system code. This requires both isolation at runtime as well as compact, lightweight distribution of code and associated state for booting. We next describe the spectrum of approaches meeting these goals, depicted in Figure 2.

OS Containers (Figure 2a). FreeBSD Jails [19] and Linux containers [38] both provide a lightweight mechanism to separate applications and their associated kernel policies. This is enforced via kernel support for isolated namespaces for files, processes, user accounts and other global configuration. Containers put the entire monolithic kernel in the trusted computing base, while still preventing applications from using certain functionality. Even the popular Docker container manager does not yet support isolation of root processes from each other.[1]

[1] `https://docs.docker.com/articles/security/`

Both the total number and ongoing high rate of discovery of vulnerabilities indicate that stronger isolation is highly desirable (see Table 2). An effective way to achieve this is to build applications using a *library operating system* (libOS) [10, 24] to run over the smaller trusted computing base of a simple hypervisor. This has been explored in two modern strands of work.

Picoprocesses (Figure 2b). Drawbridge [34] demonstrated that the libOS approach can scale to running Windows applications with relatively low overhead (just 16MB of working set memory). Each application runs in its own *picoprocess* on top of a hypervisor, and this technique has since been extended to running POSIX applications as well [15]. Embassies [22] refactors the web client around this model such that untrusted applications can run on the user's computer in low-level native code containers that communicate externally via the network.

Unikernels (Figure 2c). Even more specialised applications can be built by leveraging modern programming languages to build *unikernels* [25]. Single-pass compilation of application logic, configuration files and device drivers results in output of a single-address-space VM where the standard compiler toolchain has eliminated unnecessary features. This approach is most beneficial for single-purpose appliances as opposed to more complex multi-tenant services (§5).

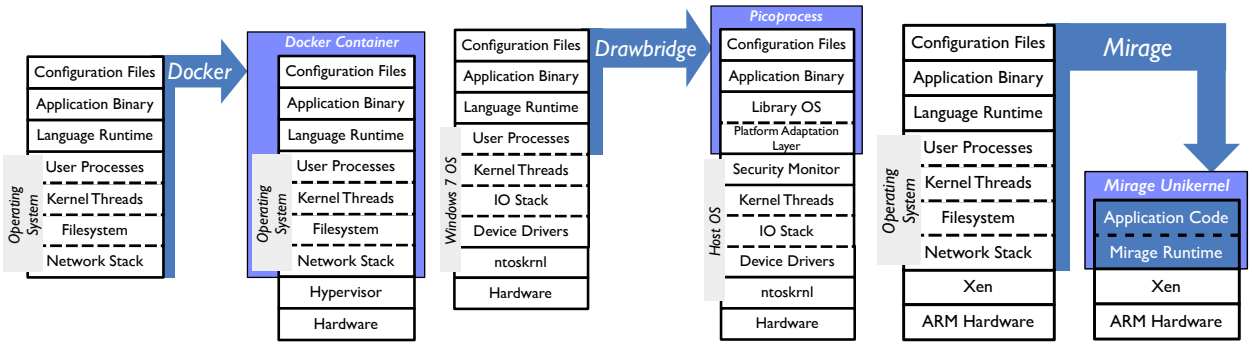
Unikernel frameworks are gaining traction for many domain-specific tasks including virtualizing network functions [29], eliminating I/O overheads [20], building distributed systems [6] and providing a minimal trust base to secure existing systems [11, 7]. In Jitsu we use the open-source MirageOS[2] written in OCaml, a statically type-safe language that has a low resource footprint and good native code compilers for both x86 and ARM. A particular advantage of using MirageOS when working with Xen is that all the toolstack libraries involved are written entirely in OCaml [36], making it easier to safely manage the flow of data through the system and to eliminate code that would otherwise add overhead [18].

2.2 ARM Hardware Virtualization

Xen is a widely deployed type-1 hypervisor that isolates multiple VMs that share hardware resources. It was originally developed for x86 processors [2], on which it now provides three execution modes for VMs: paravirtualization (PV), where the guest operating system source is directly modified; hardware emulation (HVM), where specialised virtualization instructions and paging features available in modern x86 CPUs obviate the need to modify guest OS source code; and a hybrid model (PVH) that enables paravirtualized guests to use these newer hardware features for performance.[3]

[2] `http://www.openmirage.org`

[3] See Belay et al [4] for an introduction to the newer VT-x features.



(a) Containers, e.g., Docker. (b) Picoprocesses, e.g., Drawbridge. (c) Unikernels, e.g., MirageOS.

Figure 2: Contrasting approaches to application containment.

The Xen 4.4 release added support for recent ARM architectures, specifically ARM v7-A and ARM v8-A. These include extensions that let a hypervisor manage hardware virtualized guests without the complexity of full paravirtualization. The Xen/ARM port is markedly simpler than x86 as it can avoid a range of legacy requirements: e.g., x86 VMs require `qemu` device emulation, which adds considerably to the trusted computing base [7]. Simultaneously, Xen/ARM is able to share a great deal of the mature Xen toolstack with Xen/x86, including the mechanics for specifying security policies and VM configurations.

Jitsu can thus target both Xen/ARM and Xen/x86, resulting in a consistent interface that spans a range of deployment environments, from conventional x86 server hosting environments to the more resource-constrained embedded environments with which we are particularly concerned, where ARM CPUs are commonplace.

2.3 Xen/ARM Unikernels

Bringing up MirageOS unikernels on ARM required detailed work mapping the libOS model onto the ARM architecture. We now describe booting MirageOS unikernels on ARM, their memory management requirements, and device virtualization support.

Xen Boot Library. The first generation of unikernels such as MirageOS [26, 25] (OCaml), HaLVM [11] (Haskell) and the GuestVM [32] (Java) were constructed by forking *Mini-OS*, a tiny Xen library kernel that initialises the CPU, displays console messages and allocates memory pages [39]. Over the years, Mini-OS has been directly incorporated into many other custom Xen operating systems, has had semi-POSIX compatibility bolted on, and has become part of the trusted computing base for some distributions [7]. This copying of code becomes a maintenance burden when integrating new features that get added to Mini-OS. Before porting to ARM, we therefore rearranged Mini-OS to be installed as a system library, suitable for static linking by any unikernel.[4] Functionality not required for booting was extracted into separate libraries, e.g., `libm` functionality is now provided by OpenLibM (which originates from FreeBSD's `libm`).

An important consequence of this is that a `libc` is no longer required for the core of MirageOS: *all* `libc` functionality is subsumed by pure OCaml libraries including networking, storage and unicode handling, with the exception of the rarely used floating point formatting code used by `printf`, for which we extracted code from the musl `libc`. Removing this functionality does not just benefit codesize: these embedded libraries are both security-critical (they run in the same address space as the type-safe unikernel code) and difficult to audit (they target a wide range of esoteric hardware platforms and thus require careful configuration of many compile-time options). Our refactoring thus significantly reduced the size of a unikernel's trusted computing base as well as improving portability.

Fast Booting on ARM. We then ported Mini-OS to boot against the new Xen ARM ABI. This domain building process is critical to reducing system latency, so we describe it here briefly. Xen/ARM kernels use the Linux zImage format to boot into a contiguous memory area. The Xen domain builder allocates a fresh virtual machine descriptor, assigns RAM to it and loads the kernel at the offset `0x8000` (32KB). Execution begins with the `r2` register pointing to a Flattened Device Tree (FDT). This is a similar key/value store to the one supplied by native ARM bootloaders and provides a unified tree for all further aspects of VM configuration. The FDT approach is much simpler than x86 booting, where the demands of supporting multiple modes (paravirtual, hardware-assisted and hybrids) result in configuration information being spread across virtualized BIOS, memory and Xen-specific interfaces.

[4] Our Mini-OS changes have been released back to Xen and are being integrated in the upstream distribution that will become Xen 4.6.

Some assembler code then performs basic boot tasks:

- Configuring the MMU, which handles mapping virtual to physical memory addresses.
- Turning on caching and branch prediction.
- Setting up the exception vector table, defining how to handle interrupts and deal with various faults such as reading from an invalid memory address.
- Setting up the stack pointer and jumping into the C `arch_init` function for the remainder of execution.

The early C code sets up the virtual logging console and interrupt controllers. After this, unikernel-specific C code binds interrupt handlers, memory allocators, timekeeping and grant tables [42] into the language runtime. The final step is to jump into the OCaml code section[5] and begin executing application logic. The application links memory-safe OCaml libraries to perform the remaining functions of device drivers and network stacks.

Modifying Memory Management. Once the MirageOS/ARM unikernel has booted, it runs in a single address space without context switching. However, the memory layout under ARM is significantly different from that for x86. Under the ARM Virtualization Extensions, there are two stages to converting a virtual memory address (used by application code) to a physical address in RAM, both of which go through translation tables. The first stage is under the control of the guest VM, where it maps the virtual address to what the guest believes is the physical address – the Intermediate Physical Address (IPA). The second stage, under the control of Xen, maps the IPA to the real physical address.

MirageOS' memory needs are very simple compared with traditional guest OSs. Most memory is provided directly to the managed OCaml heap which is grown on-demand. Unikernels will typically also allocate a few pages for interacting directly with Xen as these must be page-aligned and static, and so cannot be allocated on the garbage collected OCaml heap.

Although Xen does not commit to a specific fixed address for the IPA, the C code does need to run from a known location. To resolve this, the assembler boot code uses the program counter to detect where it is running and sets up a virtual-to-physical mapping that will make it appear at the expected location by adding a fixed offset to each virtual address. The table below shows this for Xen 4.5 (the latest stable release). The physical address is always at a fixed offset from the virtual address and addresses wrap around, so virtual address `0xC0400000` maps back to physical address `0` in this example.

The stack, which grows downwards, is placed at the start of RAM so that an overflow will trigger a page fault that can be caught, and can also be grown in size later in the boot process when all of the RAM is available. The 16KB translation table is an array of 4-byte entries each mapping 1MB of the virtual address space, so the 16KB table is able to map the entire 32-bit address space (4GB). Each entry can either give the physical section address directly or point to a second-level table mapping individual 4KB pages; MirageOS implements the former as this reduces possible delays due to TLB misses.

Addresses		
Virtual	Physical	Purpose
`0x400000`	`0x40000000`	Stack (16 KB)
`0x404000`	`0x40004000`	Translation tables (16 KB)
`0x408000`	`0x40008000`	Kernel image

The kernel code is followed by the `data` section containing constants and global variables, then the `bss` section with data that is initially zero and thus need not be stored in the kernel image, and finally the rest of the RAM under control of the memory allocator.

Device Virtualization. On Xen/x86 it is possible to add virtual devices by two means: pure PV devices that operate via a split-device model [42], and emulated hardware devices that use the `qemu` device emulator to provide the software model. Xen/ARM does not support the more complex hardware emulation at all, instead mandating (as a new ABI) that VMs support the Xen PV driver model to attach virtual devices.

MirageOS includes OCaml library implementations of the Xen PV protocols for networking and storage. The only modifications required from their x86 versions were the architecture-dependent memory barrier assembly instructions that differ between x86 and ARM, accessed via the OCaml foreign function interface.

The result of this work is to bring the benefits of MirageOS unikernels (compact, specialised appliances without excess baggage) to the resource-constrained ARM platform, providing an alternative to running full Linux or FreeBSD VMs. While we have described the specifics of the MirageOS port here, other teams have already picked up our work for their respective projects and are adapting it for other runtimes such as Click and Haskell.

3 The Jitsu Toolstack

We turn now to the Jitsu toolstack which supports the low-latency on-demand launching of the unikernels in response to network traffic. Our goal is to ensure that services listening on a network endpoint are always available to respond to traffic, but are otherwise not running to reduce resource utilisation. Jitsu is the Xen equivalent of the venerable `inetd` service on Unix, but instead of starting a process in response to incoming traffic, it starts a unikernel that can respond to requests on that IP address. While there have been wide-area versions of this approach in the past [1], we believe this is the first time it

[5] The `ocamlopt` compiler outputs standalone native code ARM object files that are linked with the garbage collector runtime library.

has been implemented with such low latency in a single embedded host without sacrificing isolation.

We describe Jitsu in three phases, each of which progressively reduces end-to-end latency. First, the traditional Xen toolstack is highly serialised across multiple blocking internal components, leading to large boot times due to long pauses between actual boot activity. We thus reduce these boot times by reducing this blocking behaviour and speeding up various boot components (§3.1). Jitsu preserves the existing boot protocol so that the many millions of existing Xen VM images will continue to work.

Second, we describe optimisation of the inter-VM communications protocol via *conduits*, a Plan9-like extension to support direct shared memory communication between named endpoints (§3.2). Conduits eliminate the need to use local networking to communicate between Jitsu and unikernels, further driving down latency.

Third, we introduce the *Synjitsu* directory service that masks boot latency to external clients by handling the initial stages of TCP handshake, only to hand-off the resulting state via a local conduit while the unikernel service completes booting and attaches to the network bridge (§3.3).

The net result is that a service VM can "cold boot" and respond to a TCP client in around 300–350ms, and an already-booted service can respond to local traffic in around 5ms (§4). In all cases, *all* network traffic is handled via memory-safe code in an unprivileged Xen VM.

3.1 Optimising Boot Times

Jitsu builds on the existing Xen toolstack by extending XenStore, a storage space shared between all VMs running on a physical host [12]. XenStore is a hierarchical, transactional key-value store where keys describe a path down a tree, and values store configuration and live status information for domains. Each running domain on a Xen instance has its own subtree, and so communication between domains can be coordinated via XenStore.

There are several stages to a VM booting that are triggered by XenStore: (*i*) a domain builder process loads the guest kernel image and configures it within a Xen datastructure before launching it; (*ii*) the new VM boots and attaches to its virtual devices, most notably a logging console and a network device; (*iii*) the remote end of the network and console device rings are attached to the backends that bridge them; and finally, (*iv*) the userspace starts and applications begin serving traffic.

Jitsu's utility relies on the ability to launch new VMs very quickly. Using the vanilla Xen toolstack, VM boot times are far too high for this, typically 3–5 seconds with high CPU usage for a Linux VM — hardly "just in time" when trying to start a network service with imperceptible client delay. Jitsu applies three optimisations to significantly reduce this, achieving lowest latency when booting a specialised unikernel instead of a generic VM.

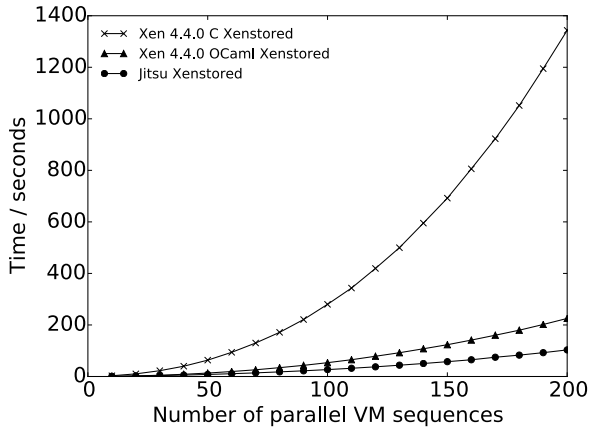


Figure 3: Comparison of different transaction reconciliation implementations during VM start/stop.

(*i*) **Domain building**. Xen's domain builder creates the initial VM kernel image. Most of its work is to initialise and zero out physical memory pages, thus guests with less memory are naturally built more quickly. As unikernels require such small amounts of memory to boot (8MB is plenty), they have an advantage over modern Linux distributions which typically require at least 64MB and are often recommended 128MB or more.

(*ii*) **Parallel device attachment**. While modern Linux parallelises much of its boot process, individual devices still have a serialisation overhead. The console device, for example, attaches to a dom0 `xenconsoled` service that drains the VM output and logs it. More significantly, attaching the network driver requires the backend domain to create a `vif` device in dom0, and to add it to a network bridge so that it can receive traffic. This blocks the VM while a slew of RPCs go back-and-forth between it and dom0, where hotplug shell scripts are executed.

This can be further sped up by parallelising the entire device attachment cycle with the domain builder itself. Jitsu starts the `vif` creation process before the domain builder runs, resulting in the two running in parallel. Although we could eliminate this overhead entirely by pre-creating domains and attaching them to the bridge (making VM launch simply a matter of attaching a unikernel to a domain before unpausing it), we prefer not to pay the cost of increased memory usage that would result from the pre-created domains.

(*iii*) **Transaction Deserialisation**. As the domain is built, a series of XenStore operations coordinates the multiple components involved in booting a VM. Building just one domain involves many transactional operations, and it becomes a latency bottleneck if they do

not parallelise well. There are two XenStore implementations provided by upstream Xen: the default is a C implementation with filesystem-based transactions, and the other an alternative written in OCaml that uses in-memory transactions with merge functions that reduce the number of conflicts [12]. We further improved the OCaml XenStore transaction handling in a Jitsu-specific fork by providing a custom merge function that handles common directory roots in parallel transactions.

Figure 3 shows the dramatic differences in VM start time when doing VM start/stop operations in parallel, with the OCaml implementations clearly more efficient than the default dæmon in C. This is due to the reduced number of conflicts which otherwise cause the toolstack to cancel and retry a large set of domain building RPCs.

Figure 4 breaks down the impact of the domain creation optimisations. The test builds the VM image with a console and network interface and starts it. As this measures only VM construction time, not boot time, it applies to both unikernels and Linux VMs. Memory usage is a significant factor in domain creation, with a 256MB domain taking a full second to create, and a 16MB domain (suitable for a unikernel) still taking a significant 650ms. Rewriting the networking hotplug scripts to use the lightweight `dash` rather than the default `bash` reduces boot time to 300ms, and eliminating forking by invoking `ioctl` calls directly rather than running shell scripts further reduces boot time to 200ms. The final two optimisations to parallelise `vif` setup and asynchronously attach the console give the end result of 120ms to boot on ARM.

Jitsu is fully compatible with x86 as well as ARM, and so we ran the same tests on a 2.4GHz quad-core AMD x86_64 server to compare boot times against ARM. The most optimised VM creation time was just 20ms on x86 – around 6 times faster than the lower powered ARM board. Although we are focused on embedded deployments in this paper, it is worth noting that such fast boot times are possible in situations where power consumption is less of a concern (§4).

3.2 Communication Conduits

Coordinating a set of running unikernels requires some means to communicate between them. For conventional VMs, all such communication passes via shared memory rings to real hardware running in a privileged VM [42]. Device-specific RPC protocols are built over these rings to provide traditional abstractions such as `netfront` (network cards) or `blkfront` (mass storage).

This is a convenient abstraction when virtualizing existing OS kernels to run under Xen, as each protocol fits into the existing device driver framework. However, the lack of user/kernel space divide in a unikernel means that it links in device drivers as normal libraries: there is no need to fit the protocols into any existing abstraction. It becomes easy to construct custom RPC layers for communication between unikernels, whether instantiated as VMs on Xen or as Linux processes.

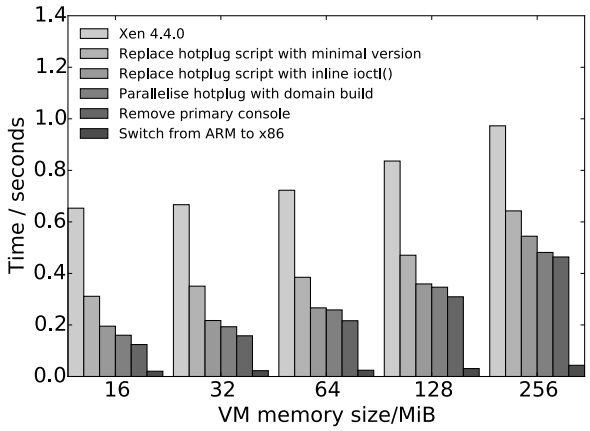


Figure 4: Optimising Xen/ARM domain build times.

Jitsu provides an abstraction over such a shared-memory communication protocol called *Conduit*, which (*i*) establishes shared-memory pages for zero-copy communication between peers; (*ii*) provides a rendezvous facility for VMs to discover named peers; and (*iii*) hooks into higher level name services like DNS. Conduit is designed to be compatible with the `vchan` library for inter-VM communication.[6]

3.2.1 Establishing a fast point-to-point connection

A `vchan` is a point-to-point link that uses Xen grant tables to map shared memory pages between two VMs, using Xen event channels to synchronise access to these pages. Establishing a `vchan` between two VMs requires each side to know its peer's domain id before the shared memory connection can be established. This allows `vchan` to work early in Xen's bootcycle before XenStore is available (e.g., within a disaggregated system [7]). Unlike previous inter-VM communication proposals [43], `vchan` remains simple by not mandating any rendezvous mechanism across VMs, focusing solely on providing a fast shared memory datapath.

Modern Linux kernels provide userspace access to grant mappings (`/dev/gntmap`) and event channels (`/dev/evtchn`), so we implemented the `vchan` protocol in pure OCaml using these devices. This required fixing several bugs in upstream Linux arising from the many ways to deadlock the system when interacting between user and kernel space. The lack of such a divide in unikernels made implementing this protocol for Mi-

[6] vchan was introduced by Qubes OS and later upstreamed to Xen; `http://openmirage.org/blog/introducing-vchan`

rageOS far simpler. The resulting code allows unikernel and Linux VMs on the same host to communicate without the overhead of a local network bridge.

3.2.2 Listening on named endpoints

For convenience, Conduit provides a higher-level rendezvous interface above `vchan` by using the existing XenStore metadata store. It extends the XenStore namespace in two places: the existing `/local/domain` tree for per-VM metadata, and a new `/conduit` tree for registering endpoint names and tracking established flows. Figure 5 shows a XenStore fragment with an HTTP client VM connecting to a HTTP server. When the server VM boots, it registers a name mapping from its domain id to `/conduit/http_server`. It then watches the `listen` key for any incoming connections.[7] The client VM similarly registers `/conduit/http_client` when it starts.

The `http_client` picks a unique port name and attempts to "resolve" the `http_server` target by writing the port name to the `listen` queue for the server (e.g., `/conduit/http_server/listen/conn1`). The server VM receives a watch event and reads the remote domain id and port name from its listen queue, giving it sufficient information to establish a `vchan`. The connection metadata is written into `/local/domain/<domid>/vchan`, and contains the grant table and event channel references through which both sides obtain their shared memory pages and virtual interrupts. The server also updates the `/flows` table with extra metadata such as per-flow statistics that can be read by management tools.

3.2.3 Access control and transactions

XenStore already has an access control model that allows per-domain access control over keys and their child nodes. This is a good fit for Conduit except during initial setup where the client domain must write directly into the `listen` directory published by the server. Although the directory is open for writing from any other VM, new keys must be restricted to only be readable by the directory owner and the creator of the key. This is analogous to setting the `setgid` and sticky bits in POSIX filesystems. With this extension added to XenStore, domains cannot observe or interfere with the creation of conduits that do not concern them, and only XenStore itself is required for rendezvous.

As XenStore is already a filesystem-like interface, this protocol is similar to the Plan 9 network model [33], with a few notable differences: (*i*) although connection establishment goes through XenStore, established channels are zero-copy shared memory endpoints that no longer require any interaction with XenStore; and (*ii*) XenStore

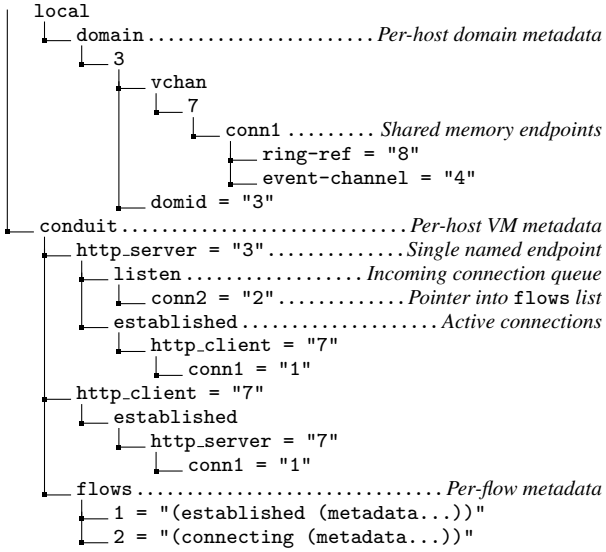


Figure 5: The XenStore tree layout for coordinating the establishment of inter-VM shared memory channels.

provides a transactional interface to let batch updates be committed atomically [12]. This eliminates potential inconsistencies arising from having state metadata spread over several keys (such as `/conduit/flows/1` and `/local/domain/3/vchan` in Figure 5).

The Conduit interface enables us to write unikernel code without having to know in advance where the remote peer is running. In the example above, the `http_server` might be a Xen unikernel or a normal Linux guest VM listening from a userspace Unix binary. Unikernels also need not trust each other as they act as a distributed system on a single host [3], communicating via a bytestream rather than directly sharing pointers into each other's address spaces.[8]

3.3 The Jitsu Directory Service

Our goal is to ensure that unikernels are launched and halted in real-time in response to network requests. This role is similar to that performed by `inetd` on Unix, and is fulfilled by the Jitsu Directory Service that maps external DNS [31] requests onto unikernel instances. When the unikernel for a service has launched, it can serve as many requests as a single VM can handle – we typically launch a VM per registered service, not one per TCP connection.

A Jitsu VM is launched at boot time with access to the external network and handles name resolution, invoked either by a local unikernel over a conduit, or through DNS protocol handlers listening on the network bridge. In the former case, the Jitsu resolver is discovered via a well-known `jitsud` Conduit node, while in the latter it

[7] A *watch* is the XenStore term for registering for notification callbacks whenever any key or value in a watched subtree is modified.

[8] The J-Kernel [40] and FlowCaml [37] provide a guide as to how pointer sharing could be safely built into future revisions of MirageOS.

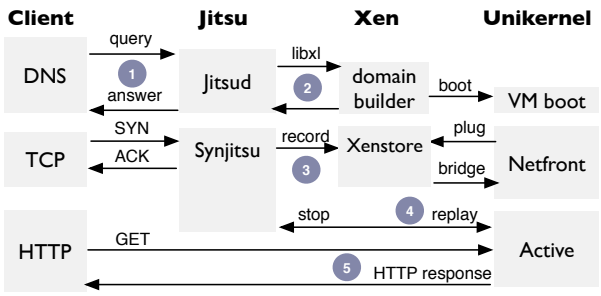


Figure 6: How Jitsu masks boot latency. ❶ DNS request triggers unikernel launch; ❷ response sent when domain building completes but before networking is active; ❸ TCP requests are buffered into XenStore until bridging is setup; and ❹ the active unikernel replays the buffered connections before ❺ directly serving traffic.

is discovered through the usual process in DNS (e.g., resolving `ns.domain.name`). If a name resolution request is received that maps onto a running unikernel, Jitsu just returns an appropriate IP address or `vchan` endpoint.

If the name requested does *not* correspond to a running unikernel, Jitsu launches the desired unikernel while simultaneously returning an appropriate endpoint (again, IP address or `vchan`) against which the client can start the higher level protocol interaction (e.g., a TCP three-way handshake). However, while the VM is starting it will not be ready to respond to network traffic as the network bridging subsystem connects asynchronously. This opens a race condition where the DNS response has been sent to the client, but the unikernel is not yet listening for the TCP `SYN` packet that will follow (likely very quickly as the client is typically local). The `SYN` packet is dropped, and the client retransmits after 1s – well outside our low-latency requirement.

3.3.1 Connection proxying via Synjitsu

We could remove this race condition by delaying the initial DNS response until the unikernel network is fully established. Instead we take advantage of the high-level libOS network stack available to us to provide a lower latency solution: we explicitly handle incoming connections in a proxy unikernel, and hand off the state to the full unikernel once it has finished plugging its network device in. This helps Jitsu to mask any latency associated with booting the target unikernel, as well as making it more robust in the face of TCP connections arriving unexpectedly outside of DNS resolution (e.g., because a client did not respect the TTL in a DNS response and attempted to connect to the service directly).

Figure 6 shows the packet flow with the `synjitsu` unikernel performing this connection proxying. When a DNS request comes in, the unikernel boot process starts,

```
conduit
└─ http_server = "3"
   └─ tcpv4 ..... Connection state from proxy
      ├─ 1 ..... Received SYN but not responded
      │  ├─ state = "SYN"
      │  └─ tcb = "(src ...)(port ...)"
      └─ 2 ..... Established and buffering packets
         ├─ state = "SYN_ACK"
         ├─ tcb = "((port ...)(isn ...))"
         └─ packets = "((data ...)(...))"
```

Figure 7: The `synjitsu` proxy registers embryonic TCP connections to mask unikernel startup time.

returning a DNS response as soon as the VM resource allocation is complete (resource exhaustion can thus be returned in the DNS response as a `SERVFAIL` to indicate the client should go elsewhere). As unikernel boot (20ms on x86, 350ms on ARM) takes longer than the RTT of a packet on a local network (5ms), it is likely that a TCP SYN would follow and be lost before the unikernel has booted, triggering a slow TCP client retransmission.

`synjitsu`, built using the same OCaml TCP stack as the booting unikernel, removes this race entirely by listening on the external network bridge and an internal conduit for TCP packets destined for a unikernel that is still booting. When it receives a `SYN`, it writes entries into a special area in the conduit XenStore tree for the booting unikernel. Figure 7 shows two examples; (*i*) where a `SYN` has been received but not responded to, and (*ii*) where a `SYN_ACK` has been sent by the proxy and the TCP data stream buffered up. When the unikernel finishes booting and has an active network interface, it signals to `synjitsu` that it is ready for traffic via a two-phase commit in XenStore, ensuring only one of them ever handles any given packet. The unikernel then reconstructs the TCP state descriptors based on the recorded state, and handles subsequent traffic on the bridge directly, with no further interference from `synjitsu`.

Splitting state across a dormant kernel and a proxy is not a new technique [1], but the high-level nature of the OCaml TCP/IP stack makes implementation a simple matter of (de)serialising values across XenStore. As only one of `synjitsu` or the unikernel ever replies to a packet, we avoid the complexity and latency increase from building a distributed network stack [16] within the host. It is also relatively easy to extend to higher-level protocols such as SSL/TLS [30], e.g., to perform the 7-way initial key exchange in one VM before it hands off the connection to another unikernel that has no access to the private keys for the remainder of its lifetime.

3.3.2 Service Configuration

Consider a client wishing to access one of a set of low-traffic websites, such as a set of personal homepages and photographs. Hosting each of these relatively low traffic

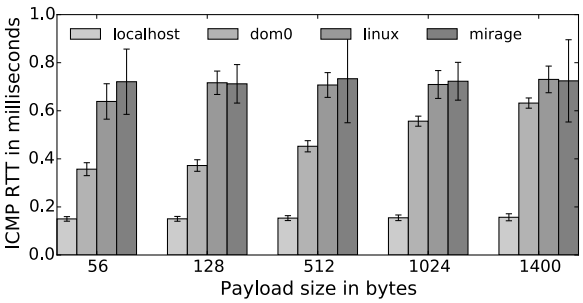


Figure 8: ICMP RTT showing the datapath latency.

Power Usage (W, 5V)		Board Model
Idle	Spinning	and active components
1.43	2.61	Cubieboard2
2.10	2.58	Cubieboard2 +Ethernet
3.36	4.49	Cubieboard2 +SSD
4.06	4.51	Cubieboard2 +SSD+Ethernet
1.72	2.86	Cubietruck
2.58	3.76	Cubietruck +Ethernet
3.92	5.51	Cubietruck +SSD
4.91	6.26	Cubietruck +SSD+Ethernet
6.84	27.02	*Intel Haswell NUC [35]*

Table 1: Power usage of the ARM boards when running Xen, with reported Intel results for comparison.

sites in the cloud would be a waste of money, while a typical small home router or similar is unlikely to have sufficient resources to keep them all simultaneously live yet isolated. An ARM device using the Jitsu toolstack is registered in the public DNS as `ns.family.name`, the nameserver for the family.name zone. When a DNS request comes in for `alice.family.name`, Jitsu returns the local external IP address configured for Alice's unikernel and performs connection proxying while Alice's unikernel launches. Conventional failover models are supported – multiple ARM boards could be registered in the DNS and return `SERVFAIL` responses if they do not have resources to serve the traffic.

In our current implementation, the Jitsu services are statically configured via OCaml code to map their unikernel with an IP address, protocol and port. We expose publication of running services via the DNS, as either an authoritative server or recursive resolver. More dynamic configurations, where launched unikernels may themselves alter the name–address–unikernel mappings and can publish using, e.g., Dynamic DNS are possible to build over this lower-level interface.

4 Evaluation

Our tests are conducted on two inexpensive off-the-shelf ARM boards: a Cubieboard2 (dual-core Allwinner ARM A20, 1GB RAM, 100Mb Ethernet) and a Cubietruck (same CPU, 2GB RAM, 1Gb Ethernet) running Xen 4.4 and an Ubuntu 14.04 dom0.[9] Our evaluation aims to answer the following questions:

- Does the port to the ARM architecture have reasonable performance, latency and energy efficiency?
- Is launching services in isolated Xen VMs a viable alternative to other approaches, e.g., containers?
- Is there any benefit in the extra isolation afforded by a type-1 hypervisor?

Throughput. We have previously carried out a fuller analysis of unikernel throughput with various protocols [25], so here we are simply verifying that there is no regression on ARM. As the ARM CPUs are considerably underpowered compared to x86 CPUs, we built a HTTP persistent queue service in MirageOS to ensure that network throughput remains acceptable. The working set of this service is larger than available RAM, and so it is served from disk. After some optimisations,[10] it served HTTP traffic at a rate of 57.92Mb/s, at which point it becomes disk bound. An `iperf` test with checksum offloading enabled revealed the same performance for Linux and MirageOS VMs.

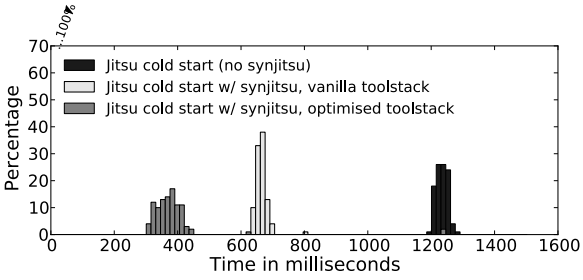
Datapath latency. The imposition of Xen and type-safety risks introducing additional latency in the network datapath, and so Jitsu minimises excess bridging (§3.2) and proxying on the data plane (§3.3.1). Figure 8 plots ICMP latency when pinging the client's own external interface (i.e., the latency of the client stack), the Xen dom0, a Linux Xen/ARM VM and a type-safe MirageOS unikernel VM. The latency difference between a Linux and MirageOS VM is never more than 0.4ms, although MirageOS does have slightly more variation.

Service Startup Latency. Figure 9a shows the end-to-end latency of HTTP requests from an external network client. First we measure the time for a "cold start" when no unikernel was running and so one had to be started by Jitsu. Early `SYN` packets are lost and the client (running Linux) retransmits them, leading to response times of over a second. We then show the effects of running `synjitsu` to proxy connection setup by intercepting `SYN` packets and handing them over to the unikernel, and also the effect of the toolstack optimisations to improve VM creation time (§3.1). Finally the latency for an already-running service is imperceptible as expected. We do not plot the start time of a full Ubuntu Linux VM, since it took over 5s with the default distribution image.

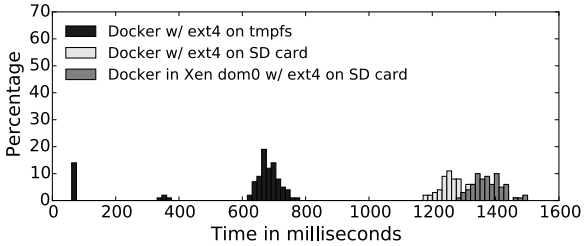
We also tested Docker 1.2.0 Linux container startup triggered from `inetd` to compare its latency with VMs. A container's start latency on a Cubieboard2 is dominated by disk I/O (Figure 9b). When running directly from a 10MB/s SD card, Docker takes at least 1.1s (na-

[9] SD card images are found at `http://blobs.openmirage.org`

[10] For full details on the profiling, see `http://bit.ly/Y3kuun`



(a) Using Jitsu from a cold start (booting a new unikernel) without Synjitsu, with Synjitsu and the ordinary toolstack, and with Synjitsu and the optimised toolstack.



(b) Using Docker: direct on Linux with RAM filesystem (`tmpfs`) and SD card, and on Xen in dom0.

Figure 9: HTTP request response times for Jitsu and Docker.

tive Linux) or 1.2s (under Xen) to spawn a new container in response to a request. To understand the effect of slow storage on Docker's start time, we also mounted Docker's volumes on an `ext4` loopback volume inside of a `tmpfs`.[11] In this configuration, container start times remained at 600ms or higher, considerably higher than Jitsu unikernels. This configuration also generated buffer IO, `ext4` and VFS errors in a significant fraction of tests resulting in early process termination.

Power Usage. A key facet of our contribution is that by using ARM-based devices, power consumption is significantly reduced, to the extent that they become acceptable to run 24/7 in a domestic environment. Table 1 shows the power usage of various configurations of our evaluation boards, measured using a custom power measurement unit we built that intercepts the USB power link to the boards. We measured each board when idle (just running Xen and a dom0), spinning in a busy loop and with Ethernet and an external solid-state drive. The SSD almost doubled power usage, but the small binary size of unikernels (around 1MB) means that in many cases we do not require a lot of space beyond that provided by the internal MMC flash. We failed to find an equivalent Intel board in the same price/performance/functionality range as the Cubieboard, and so we report power usage on the Intel Haswell NUC [35]. We also powered a Cubieboard with a USB battery unit that ran for 9 hours while logging the date every minute.

Security. To evaluate the end-to-end security properties of the Jitsu design *vs* a more conventional Linux embedded system, we looked for critical security bugs eliminated by use of (*i*) isolation via a type-1 hypervisor and (*ii*) a memory-safe language to build minimal VM appliances. Table 2 compares a recent representative selection of CVE vulnerabilities against embedded network devices (top), the Linux kernel (middle), and Xen on ARM (bottom). With Jitsu, the top group would be entirely eliminated and the middle group largely eliminated, while the bottom group would remain.

The commonest vulnerabilities still arise from protocol parsers written in unsafe languages, resulting in remote code execution vulnerabilities across the spectrum of almost every common protocol found on edge routers. Jitsu ensures that *all* traffic parsed on the external network be done so in memory-safe OCaml, mitigating this class of overflows. Another recent non-buffer-overflow vulnerability of note is *ShellShock*, a recent parsing error in the `bash` shell (CVE-2014-6271) that permits remote code execution by manipulating environment variables. The unikernel design does not include a shell, and our latency optimisations in Jitsu (§3.1) also eliminate shell scripts from the security-critical management toolstack.

The middle stream of vulnerabilities that affect the Linux kernel motivate the use of a type-1 hypervisor like Xen rather than Linux containers. Only a few bugs that affect physical device drivers can harm Xen, and even those can be mitigated in future revisions of Jitsu via driver domains [7]. The bottom stream of vulnerabilities show the class of errors that have affected Xen/ARM since its first release, and none of these are exploitable remotely. Many of these are a result of the relatively immature Xen/ARM port which has seen just one public release to date. The simplicity of the Xen/ARM codebase compared to x86 may lend itself to formal specification and verification in the future [21].

More broadly, by enabling strong isolation *inside* embedded devices, new distributed system designs leveraging multi-tenancy and low latency are possible. Systems designed to take advantage of Jitsu's isolation properties protect themselves from many passive and active attacks on wide-area network links by transmitting less data over those links and using them only for hardened, general-purpose software distribution.

[11] This rather complex configuration was required as the device-mapper in Linux 3.16 does not work directly over `tmpfs` mounts

	CVE	Description	App	Remote	Execute	DoS	Exposure	Jitsu
Embedded systems	CVE-2011-3992	SSH overflow	✓	✓	✓	✓	✓	
	CVE-2012-1800	DCP overflow	✓	✓	✓	✓	✓	
	CVE-2013-0659	UDP overflow	✓	✓	✓	✓	✓	
	CVE-2013-1605	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2013-2338	SSO overflow	✓	✓	✓	✓	✓	
	CVE-2013-4977	RTSP overflow	✓	✓	✓	✓	✓	
	CVE-2013-4980	RTSP overflow	✓	✓	✓	✓	✓	
	CVE-2013-6343	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2014-0355	HTTP overflow	✓	✓	✓	✓	✓	
	CVE-2014-3936	HNAP overflow	✓	✓	✓	✓	✓	
Linux	CVE-2014-0077	KVM overflow			✓	✓	✓	
	CVE-2014-0100	IP fragmentation		✓		✓		
	CVE-2014-0155	KVM IOAPIC				✓		
	CVE-2014-0206	AIO kernel mem					✓	
	CVE-2014-1690	IRC netfilter	✓	✓			✓	
	CVE-2014-2309	IPv6 routing mem		✓		✓		
	CVE-2014-2672	Atheros WLAN DoS		✓		✓		✓
	CVE-2014-2706	MAC 802.11 race		✓		✓		✓
	CVE-2014-5206	MNT_NS bypass					✓	
	CVE-2014-5207	MNT_NS remount				✓	✓	
Xen	CVE-2014-2580	Net disable mutex				✓		✓
	CVE-2014-2915	Processor control				✓		✓
	CVE-2014-2986	NULL deref in VGIC				✓		✓
	CVE-2014-3125	Timer context switch				✓		✓
	CVE-2014-3714	Kernel load overflow				✓	✓	✓
	CVE-2014-3715	DTB append				✓	✓	✓
	CVE-2014-3716	DTB alignment				✓		✓
	CVE-2014-3717	Kernel load overflow				✓	✓	✓
	CVE-2014-3969	Vmem privs			✓	✓	✓	✓
	CVE-2014-4021	Dirty recovery					✓	✓
	CVE-2014-4022	Dirty init					✓	✓
	CVE-2014-5147	32-bit traps				✓		✓

Table 2: A representative selection of vulnerabilities in three key system components. **App** indicates an application vulnerability. **Remote** indicates remote exploitation potential. **Execute** indicates arbitrary code execution. **DoS** indicates denial of service potential. **Exposure** indicates data exfiltration potential. **Jitsu** indicates vulnerabilities that could affect a Jitsu system (Xen on ARM with a Linux Dom0 for network drivers).

5 Discussion

Jitsu solves the problems of supporting low-latency deployment of code requiring strong isolation to resource-constrained embedded platforms. Although we focus on use of MirageOS unikernels in that specific problem domain, the techniques embodied within Jitsu have a number of attendant benefits which we discuss here.

General Jitsu. Although we have focused in this paper on using Jitsu with unikernels, we have not made any changes to the Xen guest ABI. As a result Jitsu works as described with legacy VMs (e.g., Linux, FreeBSD) on both ARM and in traditional x86 datacenter environments. This contrasts with systems such as ClickOS [28] which modifies the ABI to achieve very dense, highly parallel deployments of 10,000s of VMs in an x86_64 datacenter. We anticipate that both of these approaches will converge in upstream Xen in the future through a revision of the XenStore protocol. The one thing that Jitsu cannot provide with legacy VMs is guaranteed latency, due to the inherent boot overheads of such VMs. Tests on x86 (Figure 4) point to the intriguing possibility of very fast 20–30ms response times in datacenter environments as well.

Jitsu can easily be extended to support other VM lifecycle operations such as live relocation or VM forking [41, 23] in response to network requests. However, Jitsu is particularly well-suited to Xen/ARM through its use of explicit state transfer in `synjitsu` rather than depending on these hypervisor-level features. Forking or migrating entire VMs is more resource intensive than protocol state transfer, and is not yet fully supported by Xen/ARM 4.5. Simple TCP connection handover as in `synjitsu` is also easily extensible, and we are currently applying it to a full seven packet SSL/TLS handshake to support encrypted connections [30].

Finally, as noted previously, use of the Conduit stack for coordinating communication between VMs is not limited to unikernels. The basic principle of providing a name-based resolver to shared memory endpoints that does not depend on either a network (e.g., TCP/IP) or a process model (e.g., SysV `shmem`) can be used to interface conventional VM software stacks with unikernels.

Modularity. Jitsu both exploits and enables extensive use of modularity, which is very useful in building reliable distributed systems. The first form of modularity is found the way MirageOS is implemented – a set of lightweight OCaml libraries fulfilling module type signatures. Type-checking these signatures makes it easy to ensure that, when picking and choosing the features to be included in a particular unikernel, the basic system requirements are satisfied. As a result, developing features such as Conduit (§3.2) was far more straightforward than would have been for a traditional OS: during development it never crashed the Mini-OS kernel, and almost every error was caught and turned into an explicit condition or a high-level OCaml exception. Similarly, the `synjitsu` proxy uses the same OCaml TCP/IP library as found in the unikernels, simply with very different runtime policies.

The new Conduit capability also directly addresses one of the key criticisms of the MirageOS approach: lack of multilingual support through the dependence on OCaml. With Jitsu – specifically the combination of Synjitsu and Conduit's low latency high throughput inter-VM communication – it is entirely feasible to launch a TCP/IP MirageOS unikernel that will proxy incoming traffic to another unikernel (e.g., in Ruby or PHP) that need only implement the Conduit protocol and so need not expose, or even include, a TCP/IP implementation.

Use cases. We envisage Jitsu being useful in a wide range of situations. For example, where legacy software that may be difficult to upgrade (e.g., embedded device firmware) must be run, Jitsu can be used to provide a very narrow, application specific firewall that can filter and groom incoming traffic from the public Internet limiting the exposure of the legacy software.

Another useful scenario would be to contain application code that would normally run as a cloud service so that it can be run on a platform, such as the home router, inside the home. For example, consider the latency-sensitive applications noted earlier, Google Glass and Apple's Siri. By implementing the cloud services that support these applications as unikernels, they could be downloaded to run locally on the home router, providing significantly lower latency for common operations while still having the full power of the cloud at their disposal.

Yet other application scenarios include those where the data to be processed by the cloud-hosted service might be considered particularly personal, such as a family's photos. Photos might be hosted encrypted on the home router, and then unikernel versions of services such as Apple's iPhoto and Google's Picasa might be instantiated on-demand on the home router and given access to decryption keys held locally. Access to photos is then more directly controlled within the home without giving up all the personal data to the cloud providers [14].

Experimental artefacts. Finally, we wish to encourage use of Jitsu by other groups to explore the possibilities inherent in the platform. To that end we have made available all the code used in this paper:

- MirageOS and Jitsu are hosted on GitHub (`github.com/mirage`) with documentation on our self-hosted website at `openmirage.org`.
- The Xen/ARM and dom0 Linux distributions can be built via scripts at `github.com/mirage/xen-arm-builder`; and prebuilt SD Card images are also available for download there.
- Enquiries can be directed to our Xen.org mailing list linked from `openmirage.org/about/`.

6 Conclusions

We have presented *Jitsu*, a low latency toolstack for Xen/ARM that uses memory-safe unikernels to serve applications with significantly greater levels of isolation and security than currently achieved on modern embedded devices. Jitsu includes optimisations of the toolstack of Xen, a full-featured widely-deployed modern hypervisor that now supports ARM devices, maintaining full ABI compatibility for existing deployments. Jitsu adds the convenience of an `inetd`-like service that leverages our reduced boot latencies of around 350ms on ARM and 30ms on x86 to summon VMs in response to network traffic. Our *Synjitsu* service masks even that latency by minimally proxying connection setup requests to enable instantaneous response for clients while the unikernel boots.

The full source code is available under a BSD license at `openmirage.org`, along with documentation and installation instructions for use with Cubieboards. We welcome any patches, success stories and reports of improbable stunts conducted using Jitsu.

7 Acknowledgements

We thank Mark Shinwell, Hannes Mehnert, Raphael Proust, Malte Schwarzkopf, Matt Grosvenor, Ionel Gog, Hajime Tazaki and Martin Lucina, our paper shepherd Geoff Voelker and the anonymous NSDI reviewers for feedback. The research leading to these results received funding from the European Union's Seventh Framework Programme FP7/2007–2013 under the Trilogy 2 project (grant agreement no. 317756), and the User Centric Networking project, (grant agreement no. 611001), and 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.

References

[1] AGARWAL, Y., SAVAGE, S., AND GUPTA, R. Sleepserver: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *Proc. USENIX Annual Technical Conference (ATC)* (Boston, MA, 2010), USENIX Association, pp. 22–22.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, USA, 2003), ACM, pp. 164–177.

[3] BAUMANN, A., BARHAM, P., DAGAND, P., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 11–14 2009), pp. 29–44.

[4] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, 2012), USENIX, pp. 335–348.

[5] CHECKOWAY, S., MCCOY, D., KANTOR, B., ANDERSON, D., SHACHAM, H., SAVAGE, S., KOSCHER, K., CZESKIS, A., ROESNER, F., AND KOHNO, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proc. 20th USENIX Security Symposium (SEC)* (San Francisco, CA, 2011), USENIX Association, pp. 6–6.

[6] CLOUDOZER INC. Erlang on Xen, at the heart of super-elastic clouds. `http://erlangonxen.org/`.

[7] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, Oct. 23–26 2011), pp. 189–202.

[8] COSTIN, A., ZADDACH, J., FRANCILLON, A., AND BALZAROTTI, D. A large-scale analysis of the security of embedded firmwares. In *Proc. 23rd USENIX Security Symposium (SEC)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 95–110.

[9] ELPHINSTONE, K., AND HEISER, G. From L3 to seL4 what have we learnt in 20 years of L4 microkernels? In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farminton, Pennsylvania, 2013), ACM, pp. 133–150.

[10] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, JR., J. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, Colorado, USA, 1995), ACM, pp. 251–266.

[11] GALOIS INC. The Haskell Lightweight Virtual Machine (HALVM) source archive. `https://github.com/GaloisInc/HaLVM`.

[12] GAZAGNAIRE, T., AND HANQUEZ, V. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. *SIGPLAN Notices 44*, 9 (Aug. 2009), 203–214.

[13] GRANZER, W., PRAUS, F., AND KASTNER, W. Security in building automation systems. *IEEE Trans. Industrial Electronics 57*, 11 (Nov. 2010), 3622–3630.

[14] HADDADI, H., HOWARD, H., CHAUDHRY, A., CROWCROFT, J., MADHAVAPEDDY, A., AND MORTIER, R. Personal data: Thinking inside the box. Tech. Rep. abs/1501.04737, arXiv, Jan. 2015.

[15] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run POSIX apps in a minimal picoprocess. In *Proc. USENIX Annual Technical Conference (ATC)* (June 2013), USENIX.

[16] HRUBY, T., VOGT, D., BOS, H., AND TANENBAUM, A. S. Keep net working— on a dependable and fast networking stack. In *Proc. Dependable Systems and Networks (DSN)* (Boston, MA, June 2012).

[17] HYMAN, P. Augmented-reality glasses bring cloud security into sharp focus. *Commun. ACM 56*, 6 (June 2013), 18–20.

[18] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *Proc. USENIX Annual Technical Conference (ATC)* (San Jose, CA, 2013), USENIX, pp. 309–320.

[19] KAMP, P.-H., AND WATSON, R. N. M. Jails: Confining the omnipotent root. In *Proc. SANE Conference* (2000).

[20] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. OSv—optimizing the operating system for virtual machines. In *Proc. USENIX Annual Technical Conference (ATC)* (Philadelphia, PA, June 2014), USENIX Association, pp. 61–72.

[21] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: formal verification of an OS kernel. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, Oct. 11–14 2009), pp. 207–220.

[22] KRSUL, I., GANGULY, A., ZHANG, J., FORTES, J., AND FIGUEIREDO, R. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *Proc. ACM/IEEE Supercomputing Conference (SC)* (Nov. 2004), p. 7.

[23] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proc. 4th ACM European Conference on Computer Systems (EuroSys)* (Nuremberg, Germany, 2009), pp. 1–12.

[24] LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P. T., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications 14*, 7 (1996), 1280–1297.

[25] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: library operating systems for the cloud. In *Proc. 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, Texas, USA, 2013), ACM, pp. 461–472.

[26] MADHAVAPEDDY, A., MORTIER, R., SOHAN, R., GAZAGNAIRE, T., HAND, S., DEEGAN, T., MCAULEY, D., AND CROWCROFT, J. Turning down the LAMP: Software specialisation for the cloud. In *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (Boston, MA, USA, June 2010).

[27] MADHAVAPEDDY, A., AND SCOTT, D. Unikernels: The rise of the virtual library operating system. *Communications of the ACM (CACM) 57*, 1 (Jan. 2014), 61–69.

[28] MANCO, F., MARTINS, J., AND HUICI, F. Towards the super fluid cloud. In *Proc. ACM SIGCOMM (demo track)* (Chicago, Illinois, USA, 2014), ACM, pp. 355–356.

[29] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the art of network function virtualization. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 459–473.

[30] MEHNERT, H., AND MERSINJAK, D. K. Transport Layer Security purely in OCaml. In *Proc. ACM OCaml 2014 Workshop* (Sept. 2014).

[31] MOCKAPETRIS, P. Domain names – implementation and specification. RFC 1035 (Standard), Nov. 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

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

[33] PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. Plan 9 from Bell Labs. In *Proc. Summer UKUUG Conference* (1990), pp. 1–9.

[34] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, California, USA, 2011), ACM, pp. 291–304.

[35] S, G. T. Anandtech: Intel's Haswell NUC: D54250WYK UCFF PC review (`http://bit.ly/1Dxq6hJ`), Jan. 2014.

[36] SCOTT, D., SHARP, R., GAZAGNAIRE, T., AND MADHAVAPEDDY, A. Using functional programming within an industrial product group: perspectives and perceptions. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (Baltimore, Maryland, USA, Sept. 27–29 2010), pp. 87–92.

[37] SIMONET, V. The Flow Caml System: Documentation and user's manual. Tech. Rep. RT-0282, INRIA, July 2003.

[38] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. 2nd ACM European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal, 2007), ACM, pp. 275–287.

[39] THIBAULT, S., AND DEEGAN, T. Improving Performance by Embedding HPC Applications in Lightweight Xen Domains. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVIRT'08)* (Glasgow, Scotland, Mar. 2008).

[40] VON EICKEN, T., CHANG, C., CZAJKOWSKI, G., HAWBLITZEL, C., HU, D., AND SPOONHOWER, D. J-Kernel: A capability-based operating system for Java. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects* (1999), J. Vitek and C. D. Jensen, Eds., vol. 1603 of *Lecture Notes in Computer Science*, Springer, pp. 369–393.

[41] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)* (Brighton, United Kingdom, 2005), ACM, pp. 148–162.

[42] WARFIELD, A., FRASER, K., HAND, S., AND DEEGAN, T. Facilitating the development of soft devices. In *Proc. USENIX Annual Technical Conference (ATC)* (Apr. 10–15 2005), pp. 379–382.

[43] ZHANG, X., MCINTOSH, S., ROHATGI, P., AND GRIFFIN, J. L. XenSocket: A high-throughput interdomain transport for virtual machines. In *Proc. ACM/IFIP/USENIX International Conference on Middleware (Middleware)* (Newport Beach, California, 2007), Springer-Verlag New York, Inc., pp. 184–203.