Constructing a "Functional" Cloud (Mirage 2011)

Anil Madhavapeddy,[1] Richard Mortier,[2] Thomas Gazagnaire,[3] Raphael Proust,[1] David Scott,[4] Balraj Singh,[1]
Robert Watson,[1] Charalampos Rotsos,[1] Steven Hand,[1] Jon Crowcroft[1] and Andrew W. Moore[1]

University of Cambridge[1]
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
first.last@cl.cam.ac.uk

Horizon Digital Economy[2]
Nottingham Geospatial Building
University of Nottingham
Nottingham NG7 2TU
first.last@nottingham.ac.uk

OCamlPro SAS[3]
6, allée de la Croix Saint-Pierre
F-91190, Gif-sur-Yvette
France
first@ocamlpro.com

Citrix Systems R&D[4]
Building 101, Science Park
Cambridge CB4 0FY
United Kingdom
first.last@eu.citrix.com

In this report we present a vision of the future of Internet infrastructure and, more broadly, heterogeneous cloud computing. We take a clean-slate approach, sweeping away decades of accreted system software. We believe the advent of the latest technology discontinuity – movement of software and services into the cloud – makes this a necessary step to take, but one capable of delivering significant benefits in the long-term security, reliability and efficiency of our digital infrastructure.
The core of our approach is *specialised application synthesis*, targeting platforms ranging from virtual servers in the cloud, through desktops and servers, to mobile smartphones. We describe the initial steps we have taken toward our solution, the Mirage application synthesis framework, and finish by describing ongoing and future work. Finally, we motivate this vision by presenting several challenges that arise from different fields but have fundamental commonalities best addressed by our approach.

1. INTRODUCTION

The Internet has experienced huge growth in the last twenty five years, and hundreds of millions of users now depend on it daily for news, entertainment and commerce. Thousands of large datacenters have sprung up to fuel the demand for compute power to drive these Internet services, fundamentally changing the economics of hosting. Cloud computing (§1.1) means that software infrastructure now runs on virtual machines, and resource consumption is charged per second. Simultaneously we have seen mobile devices and social networking become popular with consumers as the way to operate on the move. Devices such as the iPhone boast fully-featured software stacks and gigabytes of solid-state storage.

We draw two commonalities from these quite different domains. First, alongside traditional metrics such as raw performance, *efficiency* is increasingly valuable. Datacenters must carefully manage energy consumption as it is both a significant operational cost, and a key constraint on the expansion of existing, and construction of new, datacenters. In the mobile case, battery technology has not kept pace at the same rate, and so an efficient software stack is vital to having a device that works for reasonable periods of time.

Second, *security* remains a serious concern on the Internet with some intrusions impacting the lives of millions of people.[1] As more software and services are moved into the cloud, they become vulnerable to attack by all-comers. At the same time, more and more personal data is being generated and stored on smartphones and tablets.

However, the evolutionary nature of the computer industry means that many software layers have built up whose security is difficult to reason about and which are not well-suited to generating small, efficient outputs. The result is an ever-increasing energy budget and system complexity, as well as a stream of serious security issues resulting from software bugs.

Our goal is to enable construction of Internet infrastructure suitable for both current and future use in the face of these trends (the move to the cloud, the ever increasing amounts of highly personal data on mobile devices). To this end we are building a clean-slate, secure programming model that changes the way we build and deploy software, particularly networked services.

[1] Sony theft: `http://www.bbc.co.uk/news/technology-13256817`

Figure 1: *A modern datacenter using cheap commodity hardware* (source: scienceblogs.com)

The key characteristic of our approach is to ease the process of application specialisation: the synthesis of binaries suitable for execution on diverse resources ranging from individual smart-phones, to sets of cloud resources. As a result, this model is well-suited to our modern energy-constrained, highly mobile society that generates vast amounts of digital data for sharing and archiving.[2]

In this paper we begin by briefly introducing cloud (§1.1) and mobile computing (§1.2) for non-experts. We then draw out the common ground between these seemingly diverse platforms (§2), and describe how we address them in Mirage (§3). Finally, we review related work (§4), discuss the current status of the project (§5), next steps (§6), and suggest one of the many interesting future possibilities we believe Mirage enables (§7).

1.1. Cloud Computing

In the late 1990s, it was common for a company needing Internet hosting to purchase physical machines in a hosting provider. As the company's popularity grew, their reliability and availability requirements also grew beyond a single provider. Sites such as Google and Amazon started building datacenters spanning the USA and Europe, growing their reliability and energy needs.

For example, in 2006 the EPA estimated that datacenters consumed 1.5% of the total electricity bill of the entire USA – around $4.5bn [10] – and there was a rush for hydro-electric power in more remote locations in the US. Companies over-provisioned to deal with peak demand (e.g., Christmas time), and had idle servers during off-peak periods. In early 2000, researchers began to examine the possibility of dividing up commodity hardware into logical chunks of compute, storage and networking which could be rented on-demand by companies with only occasional need for them. The XenoServers project forecast a network of these datacenters spread across the world [12].

Meanwhile, the Xen hypervisor was developed in Cambridge in 2003, to divide up a physical computer into multiple chunks [1]. In the following 4 years, it grew to become the leading open-source virtualization solution, and was adopted by Amazon as part of its "Elastic Computing" service.[3] In this way, Amazon became the first commercial provider of what is now dubbed "cloud computing" – a rental service of a slice of a huge datacenter to provide on-demand computation resources that can be dynamically scaled up and down.

Cloud computing has continued to prove popular in the era of Web2.0 services, which experience frequent "flash traffic" that require large amounts of resources for short periods of time – cloud computing is a natural fit for this kind of highly variable demand. Other common uses are for huge but one-off tasks, e.g., the New York Times scanning their back catalogue of 11 million articles from 1851 to 1922 into PDF format for online delivery. The process used 100 virtual machines for a 24 hour period [13]. Finally, companies with large data processing requirements have been constructing distributed systems to scale up to thousands of machines, e.g., Google's BigTable [4] and Amazon's Dynamo [7].

However, the evolutionary nature of the development of cloud computing has led to some significant economic and environmental challenges. Virtualization encapsulates an entire operating system and runs it in a virtual environment, introducing yet another layer to the already bloated modern software stack. Vinge has noted that a compatibility software layer is added every decade or so (operating systems, user processes, threading, high-level language runtimes, etc.), and that without some consolidation we are headed for a future of "software archaeology" where it will be necessary to dig down into software layers long forgotten by the programmers of the day [28].

In the context of a large datacenter, these layers are an efficiency and security disaster: they add potentially millions of lines of code performing redundant operations, and exposing countless unknown vulnerabilities to attack. It is here that we position our work: compressing all these layers at compilation time into a specialised appliance that can execute directly against the virtual hardware interface.

1.2. Mobile Handsets

Over the past decade mobile phones have evolved dramatically. Starting out as simple (and bulky!) handsets capable of making and receiving calls, sending and receiving SMS messages, and providing simple

[2] XKCD expresses this better than we can: `http://xkcd.com/676/`

[3] `http://aws.amazon.com/`

Figure 2: *Comparing a phone from a couple of decades ago with a modern Apple iPhone*

address books, they are now mobile computation devices with processors running at gigahertz speeds, many megabytes of memory, gigabytes of persistent storage, large screens supporting multi-touch input, and a wide array of sensors including light-sensors, accelerometers, and accurate global positioning system (GPS) location devices. As a result so-called "smartphones" are now capable of running multiple user-installed applications, with many opening online application stores to support users discovering, purchasing and installing applications.[4]

Devices such as the iPhone and Android phones run commodity operating systems (OSs) – iOS and Linux respectively – with selected customisations. For example, both those platforms remove page swapping from the virtual memory subsystem, instead warning and then halting entire applications when memory pressure is detected. As the persistent storage in these devices is of commensurate size and speed to the RAM, both platforms consider swapping out individual pages to have little benefit.

However, *power* remains a basic resource constraint that it is difficult to properly address when basing off commodity OSs. Battery technology for mobile handsets has not kept pace with development of their other resources such as compute, memory and storage. As a result, while very basic mobile phones have had standby lifetimes of over a week and talk-times of several hours for most of the last decade, modern smartphones fare dramatically worse. For example, battery lifetimes of a device such as the iPhone – which one might expect has been heavily optimised in this regard as the hardware and OS are under the control of a single company – are reported to be on the order of a couple of days standby and no more than a couple of hours talk-time.

Consequently, modern smartphones present a situation where efficiency is already key, and is only increasing in importance as energy-hungry capabilities increase. At the same time, we persist in running system software stacks that can trace many of their core technologies back to the days of mainframes! We thus position our work here not so much to reduce the aggregate energy demand of these devices or to improve their performance particularly, but to dramatically improve the user experience by making the devices last longer on a single battery charge.

[4]The biggest is the Apple App Store at `http://store.apple.com/`

2. APPLICATION SYNTHESIS

We claim that the basic aims of increased security and efficiency noted in previous sections can be addressed through a single approach: *specialisation* of each application, eliminating both inefficiency and attack vectors. A similar trend has been observed in other parts of the Internet's infrastructure over the years: Cisco routers, NetApp filers, Riverbed compression and IronPort email filters are all specialised systems that do one thing well at the cost of generality.

The missing piece in enabling ubiquitous specialisation is a programming framework that makes it easy to construct such appliances from reusable software and hardware components. Thus our basic approach is "application synthesis". Rather than building a single binary designed to run everywhere as, e.g., Java espouses, we claim that providing the system with more information at compile time enables it to generate highly specialised outputs that perform only a small set of tasks, but do so with greater efficiency, reliability and security.

Current software stacks, e.g., LAMP,[5] are too "thick," containing now unnecessary support for legacy systems and code. By applying developments from the last 20 years of computer science research in fields such as privacy, virtualization, and languages (both practical runtimes and theoretical underpinnings), we

[5]Linux, Apache, MySQL, PHP/Python

can dramatically simplify the software stack in use. As a result we believe we can achieve the following benefits:

Simplicity. By providing a common development toolchain and runtime environment, a single program can be easily retargeted to different platforms, ranging from mobile devices to web browsers to cloud-hosted virtual machines. This allows the programmer to focus on their program without needing to explicitly manage variations in storage, networking and other infrastructure.

Efficiency. Legacy support in modern software stacks introduces overheads, wasting energy and, as a result, money.[6] Further, services such as AWS have introduced "spot pricing," enabling consumers to bid for spare capacity, paying in proportion to demand.[7] To make efficient use of such a facility requires the ability to instantiate and destroy virtual machine images with very low latency, difficult with today's heavyweight software stacks such as LAMP.

Security. By generating a specialised image generated from code written in a strongly typed and automatically type-checked language, program security increases as unnecessary functionality is removed at compile time. Additionally, the specialisation process can even compile in configuration information, partially evaluating the result and statically rejecting invalid configurations before the system is deployed in the wild.

3. MIRAGE

Mirage is a clean-slate application synthesis framework able to transparently generate binaries for a range of backends from a single high-level source code. Supported backends are highly diverse, ranging from standalone microkernels that run directly on the Xen hypervisor, to UNIX binaries, and even Javascript executables that run in a browser.

Applications built through Mirage effectively become operating systems in their own right, managing their own resources either explicitly – e.g., running on the Xen hypervisor – or implicitly – e.g., running as a UNIX process.

Mirage applications do not need to be developed in a spartan microkernel environment: the UNIX backend enables the full suite of standard development and debugging tools to be used. When production deployment is desired for a working application, the debugged and tested code is simply recompiled as a standalone Xen microkernel, requiring no further intervention from the developer.

The specialisation process works across many layers: configuration files, source code optimisations, and also the language runtimes that interface with specific hardware. For example, Mirage applications running under Xen can make more effective use of live relocation [5], device hotplug [32], and low-energy computation modes.

We next describe how we simplify the software stack by eliminating runtime abstractions (§3.1). We also take advantage of modern programming languages to perform safety checks at compilation time, so they can be removed from running code (§3.2). Mirage *statically specializes* applications at compilation-time based on configuration variables. This eliminates unused features entirely, keeping the deployments smaller (§3.3). Finally, we review the concurrency model (§3.4).

3.1. Reducing Layers

Although virtualisation is good for consolidating under-utilized physical hosts, it comes with a cost in operational efficiency. The typical software stack is already heavy with layers: (*i*) an OS kernel; (*ii*) user processes; (*iii*) a language runtime such as the JVM or .NET CLR; and (*iv*) threads in a shared address space. Virtualization introduces yet another runtime layer which costs resources and energy to maintain, but is necessary to run existing software unmodified (Figure 3).

Explicit combination of operating system kernels (comprising millions of lines of code) and language runtimes (also typically millions of lines of code) has generally failed to gain ground due to the strong divide between "kernel" and "userspace" programming. Standards designed decades ago, such as POSIX, dictate kernel interfaces, and language runtimes are forced to use this inefficient legacy interface to access physical resources even as hardware rapidly evolved. Similarly, there are numerous models for concurrency: virtual machines, the OS kernel, processes and threads are all scheduled independently and preemptively. Modern multicore machines thus spend an increasing amount of time simply switching between tasks, instead of performing useful work. Mirage exposes a simpler concurrency model to applications (§3.4).

A modern garbage-collected application consists of two components: (*i*) a "mutator" which represents the main program; and (*ii*) a garbage collector to manage the memory heap. When executing under UNIX, an application uses the *malloc(3)* or *mmap(2)* library functions to allocate an area of memory, and the garbage collector manages all further activity within that region.

However, this can result in harmful interactions – consider a server that invokes *fork(2)* for every incoming network connection. The garbage collector, now present in multiple independent processes due to the use of fork, will sweep across memory to find and reclaim unused objects, and then compact the remaining memory. This

[6] We leave it to the reader to decide which is more important
[7] `http://cloudexchange.org/`

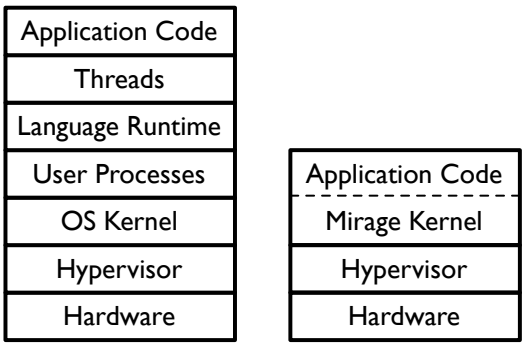

Figure 3: *A conventional software stack* (left)*, and the Mirage approach with statically-linked kernel and application* (right)

compaction touches the complete process memory map, and the kernel allocates new pages for each process' garbage collector. The result is an unnecessary increase in memory usage even when there is a lot of data shared between processes.

This is just one example of how standard operating system idioms such as copy-on-write memory can introduce inefficiency into higher level programming language implementations. As Mirage synthesises an output for each particular platform, e.g., Xen, it can eliminate these abstraction layers at compilation time, in turn reducing the work done by the system at runtime.

The inefficiency of high-level languages compared to C or Fortran has traditionally been cited as the reason they are not more widely adopted, along with the lack of mature tool-chains and good documentation [31]. Following our previous work in this area [20; 21], we intend that Mirage should close the performance barrier to C.

3.2. Functional Programming

OCaml [19] is a modern functional language based on the ML type system [18]. ML is a pragmatic type system that strikes a balance between the unsafe imperative languages (e.g., C) and pure functional languages (e.g., Haskell). It features type inference, algebraic data types, and higher-order functions, but also permits the use of references and mutable data structures – and guarantees all such side-effects are always type-safe and will never cause memory corruption. Type safety is achieved by two methods: (*i*) static checking at compilation time by performing type inference and verifying the correct use of variables; and (*ii*) dynamic bounds checking of array and string buffers.

OCaml is particularly noted among functional languages for the speed of binaries which it produces, for several reasons: (*i*) a native code compiler which directly emits optimised assembler for a number of platforms; (*ii*) type information is discarded at compile time, reducing runtime overhead; and (*iii*) a fast generational, incremental garbage collector which minimises program interruptions.

The lack of dynamic type information greatly contributes to the simplicity of the OCaml runtime libraries, and also to the lightweight memory layout of data structures. OCaml is thus a useful language to use when leveraging formal methods. It supports a variety of programming styles (e.g., functional, imperative, and object-oriented) with a well designed, theoretically-sound type system that has been developed since the 1970s. Proof assistants such as Coq [16] exist which convert precise formal specifications of algorithms into certified executables.

Mirage is written in the spirit of vertical operating systems such as Nemesis [14] or Exokernel [8], but differs in certain aspects: (*i*) apart from a small runtime, the operating system and support code (e.g., threading) is entirely written in OCaml; and (*ii*) is statically specialised at compile-time by assembling only required components, e.g., a read-only file system which fits into memory will be directly linked in as an immutable data structure.

The Mirage clean-slate approach means that we must reimplement network and storage protocols directly in OCaml, without having to re-use potentially insecure libraries written in C. Repositories of freely-available specifications such as the IETF RFCs have been essential in this effort, as Internet hosts do not generally care about the language or operating system a host runs, only that it supports the standard network protocols. Thus, the open and (mostly) end-to-end nature of the Internet has enabled the approach taken in this research.

3.3. Static Specialization

Operating systems provide various services to host applications, such as file systems, network interfaces, process scheduling and memory management. Conventional monolithic kernels include large amounts of code to support many permutations of these services on-demand, an essential approach when running varied applications such as desktop software or development environments. However, these large codebases can make bug-hunting and formal reasoning, e.g., model checking, about the complete system a daunting task [26].

Mirage takes a different approach: almost every service provided by the OS is implemented as a library written in OCaml, and can be linked in as required by the application. Thus, only features which are actually used are included in the output binary, instead of a whole kernel as with Linux. To date, we have implemented type-safe Ethernet, ARP, IPv4, TCP/IP, HTTP, DNS, SSH, OpenFlow, DHCP and SMTP, and interoperate with filesystems such as FAT for storage, along with custom non-standard protocols for experimentation.

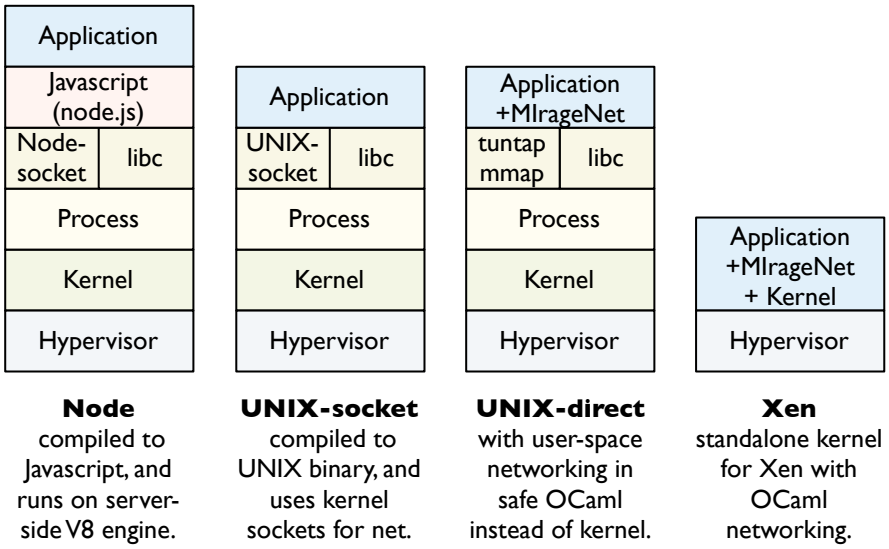

Figure 4: *Software layers for different synthesis modes in Mirage. Applications can be compiled to Javascript (for debugging or interoperability), and all the way down to microkernels. As the layers reduce, the proportion of safe code also increases, making the Xen backend the most specialised and type-safe way to deploy code.*

Of course, this does not make general purpose operating systems obsolete, but rather pushes them into a management role to spawn specialised services that are actually exposed to external traffic. This is analogous to high-performance network routers, that split these concerns into: (*i*) a general-purpose control plane; and (*ii*) a specialised data plane for high-volume traffic. The control plane is designed to be flexible to allow easy management of a network device. It exists primarily to configure the data plane, through which data flows with much less computation.

A concrete example is the `http://openmirage.org/` website, which can be synthesised into several modes described in Figure 4. A normal development build will run in UNIX userspace and bind to kernel sockets, permitting easy debugging and implementation of the website itself. It can also be recompiled to specialise its network stack using the Mirage TCP/IP implementation (in OCaml), but continue to run under UNIX. Finally, the ultimate production deployment is to relink the whole website (including its storage), into a single Xen microkernel that requires no operating system at all, and has the majority of its services implemented in safe OCaml.

As an illustration of the portability, Mirage applications can also be synthesised into Javascript, and run in the browser (within the limitations of the browser security model). Other backends we are investigating include direct hardware synthesis to FPGAs, to 16-bit PIC micro controllers,[8] and Java bytecode. The Xen microkernel backend is particularly interesting from the perspective of Internet infrastructure, as it means that critical services such as DNS could be improved by turning them into "single-shot" appliances that only perform that one service.

3.4. Concurrency

Mirage defines two distinct types of concurrency: (*i*) lightweight control-flow threads for managing I/O and timeouts; and (*ii*) parallel computation workers. This is an important distinction, since it is difficult to create a threading model which is suitable for both large-scale distributed computation and highly scalable single-server I/O [29].

Mirage I/O threads are based on the *Lwt* co-operative threading library [30] which exposes a programming abstraction similar to pre-emptive threads, but operates as an event-driven system under the hood. This is possible as OCaml is powerful enough to write the thread library in pure OCaml, minimising dependencies on the underlying platform – particularly useful when the set of underlying platforms ranges from a multi-core x86 CPU to a single-threaded Javascript runtime!

Lwt has a thread scheduler written in OCaml that can be overridden by the programmer. This opens up the possibility of user-defined policies for thread scheduling depending on the hardware platform in use. In mobile phones, for example, the scheduler could be power-aware and run some threads at a lower CPU speed, or even deliberately starve unnecessary computation in the event of low battery charge. For server operation, lightweight threading is significantly faster than using pre-emptive threads due to the removal of frequent context switching.[9]

[8] `http://bit.ly/pNoeES`

[9] `http://eigenclass.org/hiki/lightweight-threads-with-lwt`

A useful feature of Mirage threading is that the blocking nature of a function is exposed in its type, informing the programmer that a library call might take some time. Such explicit information is characteristic of statically typed systems, where the type often provides documentation to the programmer as well as safety.

For running computations that require more than one core, we adopt a parallel task model based on the join calculus [11], which is integrated directly into OCaml. The actual implementation depends on the hardware platform: (*i*) cloud platforms spawn separate VMs which use fast inter-domain communication or networking if running on different hosts; (*ii*) desktops running an operating system spawn separate user processes using shared-memory; and (*iii*) mobiles which are energy constrained remain single-threaded and use Lwt to interleave the tasks as with lightweight threads.

The use of separate lightweight virtual machines to support concurrency ensures that our model scales from multicore machines to large compute clusters of separate physical machines. It also permits the low-latency use of cloud resources which allow "bidding" for resources on-demand. A credit scheduler could thus be given financial constraints on running a parallel task across, e.g., 1000 machines at a low-usage time, or constraining it to just 10 machines during expensive periods. We have also investigated alternative programming models such as the use of dynamic dataflow graphs [24] to enable Turing-powerful, fault-tolerant distributed computation – a requirement in the world of cloud computing where hosts or networks can fail at any time.

Note that we are not trying to solve the general problem of breaking an application down into parallel chunks. Instead, we anticipate new use-cases for building distributed applications that result from our extreme specialisation. For example, an application to perform processing of images on mobile phones could be written as a single application with some threads running on mobile phones, and others running in the cloud. This way energy is not wasted on the mobile device by default, but the user can manually trigger the task to run there if desired, e.g., if they do not trust the cloud or lack connectivity, since the code is compatible. We have also speculated on the rise of "dust clouds" to improve anonymous communication on the Internet [23]. We consider two possible future applications enabled in this way in §7.

4. RELATED WORK

The Mirage operating system is structured as a "vertical operating system" in the style of Nemesis [14] and Exokernel [8]. However, in contrast to either it simultaneously targets the hardware-independent virtual cloud platform and resource-constrained mobile devices. Mirage also provides high-level abstractions for threading and storage using OCaml, making it far easier to construct applications than writing them directly in C. The theme of energy as a first-class scheduling variable was also explored in Nemesis [25].

More recently, Barrelfish [2] provides an environment specifically for multicore systems, using fast shared-memory communication between processes to work. It also incorporates support for high-level Domain Specific Languages to generate boiler-plate code for common operations [6]. Mirage aims to remove the distinction between multicore and clustered computers entirely, by adopting compiler support to use a communication model appropriate to the hardware platform's constraints.

Our use of OCaml as the general-purpose programming language has similarities to Singularity [15], which is derived from the Common Language Runtime. Singularity features software-isolated processes, contract-based communications channels, and manifest-based programs for formal verification of system properties. Mirage takes a similar approach to language-based safety, but with more emphasis on generative meta-programming [21] and a lighter statically-typed runtime. Our event-driven threading model also helps target smaller mobile devices which would struggle to run the full CLR environment.

5. STATUS

The Mirage project began in November 2009, and will be released in preview form in November 2011. Development has been, and is, entirely open-source, with documentation available through a self-hosting website,[10] and code available through GITHub.[11]

Significant milestones have been:

- March 2010: Xen and UNIX backends.
- June 2010: First prototype in USENIX HotCloud.
- September 2010: Ethernet, IPv4 and TCP/IP .
- November 2010: Amazon EC2 support.
- March 2011: Javascript backend, DNS integrated.
- May 2011: Cryptographic libs (AES, MD5, etc).
- June 2011: First Mirage summer program for students (supported by Verisign).
- July 2011: OpenFlow controller integrated.
- August 2011: Test suite for regression/perf.
- September 2011: Tutorial series at ICFP 2011.

[10] `http://openmirage.org/`
[11] `https://github.com/avsm/mirage/`

- November 2011 (upcoming): Tutorial series at CHANGE/OFELIA OpenFlowsummer school.
- November 2011 (projected): First public release, and DNSSEC integrated.

Mirage development has been iterative: we build and evaluate small features in code branches, and usually write a short blog entry on the website describing the feature. Example articles include the regular expression library,[12] the web service support,[13] and the test harness.[14]

We have also delivered talks to the community to gather feedback on approaches: at USENIX HotCloud 2010 and the ML Workshop 2010, at universities such as UCLA and Imperial College, and to industry via talks at LinkedIn and symposiums such as the Verisign 25 Years of .com.

6. NEXT STEPS

During the summer of 2011 we made significant progress in converting Mirage from a prototype into a "real" system suitable for deployment on the Internet, and a public release is projected for November 2011. Our next steps are to deploy the core system to understand its behaviour under real traffic, build a community of users, as well as continuing to develop novel applications that are only possible when using Mirage such as the two described in this section.

The research so far clearly indicates that the approach of application specialisation and synthesis can provide significant benefits for Internet infrastructure across many problem domains. We plan to deploy several network protocol implementations, such as OpenFlow (providing software defined networking), DNSSEC, SSH and HTTP and to test their interoperability and performance against software such as BIND, Open vSwitch and Apache. One challenging protocol left to design and implement is SSL, as the main implementation in use (OpenSSL) provides a rather challenging and error-prone interface to programmers.

Even unoptimised, the runtime size for a cloud backend is around 1MB in size, in contrast to Linux distributions which are difficult to squeeze below 16MB and Windows which runs into hundreds of megabytes. Thus, specialised Mirage binaries should save costs (both storage and memory) when deployed in production, and we hope this will provide an incentive for industry players to transition and help us test the system "in the wild".

Next generation Internet infrastructure *must not* be vulnerable to the same classes of security problems as the existing routers and servers that form the Internet. Mirage features a protocol stack that is written entirely in a type-safe programming language (OCaml) and thus continuing this development and providing viable alternatives to coding in C is vital to ensure the stability and safety of the Internet for the next 25 years.

Mirage also makes it *simpler* for the programmer to experiment with new protocols such as multi path TCP or SPDY, and we intend to provide a simulation backend to help with large-scale distributed system testing before putting them into deployment. The Mirage synthesis approach means that the *same* source code that runs cloud services could be simulated, often difficult to do today, particularly at reasonable performance, due to code being entangled with OS interfaces.

We are pushing the approach taken by Mirage even further down the stack, e.g., into the networking hardware itself using SR-IOV and smart NICs [22], allowing the synthesis engine to use hardware if available, and fallback to software if not. Moving up the stack, the use of OCaml as our base language provides us with a solid foundation to integrate more experimental type systems, such as dependently-typed DSLs [3] and linear typing [9]. These allow more dynamic checks to be eliminated safely at compilation-time, further improving the efficiency of runtime code without sacrificing safety properties.

7. FUTURE POSSIBILITIES

The Internet is used by millions of users to communicate and share content such as pictures and notes. Internet users spend more time at "social networking" sites than any other.[15] The largest site, Facebook, has over 500 million registered users, all sharing photos, movies and notes. Personal privacy has suffered due to the rapid growth of these sites [17] as users upload their public and private photos and movies to a few huge online hosting providers such as Google and Facebook.

These companies act as a hub in the digital economy by sharing data, but their centralized nature causes serious issues with information leaks, identity theft, history loss, and ownership. The current economics of social networking are not sustainable in the long-term without further privacy problems. Companies storing petabytes of private data cover their costs by selling access to it, and this becomes more intrusive as the costs of the tail of historical data increases over time.[16]

This dependence on commercial providers to safeguard data also has significant wider societal implications. The British Library warned "historians face a black

[12] `http://openmirage.org/wiki/ocaml-regexp`
[13] `http://openmirage.org/wiki/cow`
[14] `http://openmirage.org/wiki/performance`
[15] `http://www.nielsen-online.com/pr/pr_090713.pdf`
[16] e.g., Facebook's Misrepresentation of Beacon's Threat to Privacy: Tracking users who opt out or are not logged in, `http://bit.ly/i2s4I`

hole of lost material unless urgent action is taken to preserve websites and other digital records."[17] If Facebook suffered large-scale data loss, huge amounts of personal content would be irretrievably lost. The article notes it is impractical for a single entity to archive the volume of data produced by society.

We believe an inversion of the current centralized social networking model is required to empower users to regain control of the digital record of their lives. Rather than being forced to use online services, every individual needs their own *personal container*[18] to collect their own private data under their control. By moving data closer to its owners, it is possible to create a more sustainable way of archiving digital history similar to the archival of conventional physical memorabilia. Users can choose how to share this data securely with close friends and family, what to publish, and what to bequeath to society as a digital will.

Building personal containers using today's existing programming models is difficult. From a security perspective it is dangerous to store decades of personal digital data in a single location, without strong formal guarantees about the entire software stack on which the software is built. Simply archiving the data in one place is not as useful as distributing it securely around the myriad of personal devices and cloud resources to which we now have access. Thus our vision for the future of a lifetime of personal data management also requires a secure programming model.

Personal Containers are purpose-built distributed servers for storing large amounts of digital content pertaining to one person. They run on a variety of environments from mobile phones to desktops to the cloud. Sources feeding data into an individual's personal container can include local applications (e.g., Outlook, iPhoto), online services (e.g., Google, Facebook), or devices (e.g., real-time location data, biometrics). Unlike existing storage devices, a personal container also has built-in logic to manage this data via, e.g., email or the web. This encapsulation of data and logic is powerful, allowing users to choose how they view their own data rather than being forced to use a particular website's interface. It also ensures data never leaves the personal container unless the user requests it, an important guarantee when sensitive information such as location or medical history is involved.

The personal container network is built on the same distributed principles as email, in which no central system has overall control. Instead, individuals direct queries to their personal container, whether running on their phone, at their home, or in the cloud. We liberate our data from the control of a few large corporations. It thus becomes a matter of incentive and utility whether we choose to share our data with others, whether individuals or corporations.

However, building personal containers requires a highly efficient software stack since the economies of scale available to today's massive entities, such as Facebook, are unavailable to the individual. This also applies to security: cloud providers can hire experts to prevent intrusions into their data silos, whereas individuals who do not have the technical knowledge to keep their systems up-to-date require a better solution. Today's general purpose operating systems leave users extremely vulnerable to fast-spreading viruses [27], and so our approach of building specialised appliances will also enable more robust home infrastructure.

[17] Websites 'must be saved for history' ", Guardian, `http://www.guardian.co.uk/technology/2009/jan/25/preserving-digital-archive`

[18] `http://perscon.net/`

REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44, New York, NY, USA, 2009. ACM.

[3] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (DSLs) for network protocols (position paper). In *ICDCSW '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 208–213, Washington, DC, USA, 2009. IEEE Computer Society.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[5] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium of Networked Systems Design and Implementation*, May 2005.

[6] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *5th Workshop on Programming Languages and Operating Systems (PLOS)*. ACM, 2009.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall,

and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

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

[9] R. Ennals, R. Sharp, and A. Mycroft. Linear types for packet processing. In D. A. Schmidt, editor, *13th European Symposium on Programming (ESOP), part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, volume 2986 of *Lecture Notes in Computer Science*, pages 204–218, Barcelona, Spain, April 2004. Springer.

[10] EPA. EPA report to Congress on server and data center energy efficiency. Technical report, U.S. Environmental Protection Agency, 2007.

[11] F. L. Fessant and L. Maranget. Compiling join-patterns. *Electr. Notes Theor. Comput. Sci.*, 16(3), 1998.

[12] K. A. Fraser, S. M. Hand, T. L. Harris, I. M. Leslie, and I. A. Pratt. The XenoServer computing infrastructure. UCAM-CL-TR 552, University of Cambridge, Jan. 2003.

[13] D. Gottfrid. Self-service, prorated super computing fun! - open blog - nytimes.com. *New York Times*, November 2007.

[14] S. M. Hand. Self-paging in the Nemesis operating system. In *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 73–86, Feburary 1999.

[15] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[16] INRIA-Rocquencourt. The Coq proof assistant.

[17] B. Krishnamurthy and C. E. Wills. On the Leakage of Personally Identifiable Information Via Online Social Networks. *Second ACM SIGCOMM Workshop on Online Social Networks*, 2009.

[18] X. Leroy. The Zinc experiment: An economical implementation of the ML language. 117, INRIA, 1990.

[19] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, 2005.

[20] A. Madhavapeddy. *Creating High-Performance Statically Type-Safe Network Applications*. PhD thesis, University of Cambridge, 2007.

[21] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" Internet. *SIGOPS Oper. Syst. Rev.*, 41(3):101–114, 2007.

[22] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope. Getting 10 Gb/s from Xen: Safe and fast device access from unprivileged domains. In L. Bougé, M. Forsell, J. L. Träff, A. Streit, W. Ziegler, M. Alexander, and S. Childs, editors, *Euro-Par Workshops*, volume 4854 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2007.

[23] R. Mortier, A. Madhavapeddy, T. Hong, D. Murray, and M. Schwarzkopf. Using dust clouds to enhance anonymous communication. In *18th International Workshop on Security Protocols*, April 2010.

[24] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[25] R. Neugebauer and D. McAuley. Energy is just another resource: Energy accounting and energy pricing in the Nemesis OS. In *HotOS*, pages 67–72. IEEE Computer Society, 2001.

[26] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire Linux distribution for security violations. In *Proceedings of 21st Annual Computer Security Applications Conference (ACSAC)*, pages 13–22. IEEE Computer Society, 2005.

[27] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association.

[28] V. Vinge. *A Deepness in the Sky*. Tor Books, March 1999.

[29] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[30] J. Vouillon. Lwt: a cooperative thread library. In *ML '08: Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 3–12, New York, NY, USA, 2008. ACM.

[31] P. Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, 1998.

[32] A. Warfield, K. Fraser, S. Hand, and T. Deegan. Facilitating the development of soft devices. In *Proceedings of the 2005 USENIX Annual Technical Conference (General Track)*, pages 379–382. USENIX, April 2005.