Multiscale not Multicore: Efficient Heterogeneous Cloud Computing

Anil Madhavapeddy
University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
avsm2@cl.cam.ac.uk

Richard Mortier
University of Nottingham,
Sir Colin Campbell Building,
Nottingham NG7 2TU
United Kingdom
rmm@cs.nott.ac.uk

Jon Crowcroft
University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
jon.crowcroft@cl.cam.ac.uk

Steven Hand
University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
steven.hand@cl.cam.ac.uk

In this paper, we present a vision of the future of heterogeneous cloud computing. Ours is a clean-slate approach, sweeping away decades of accreted system software. We believe the advent of the latest technology discontinuity—the move to the virtual cloud—makes this a necessary step to take, but one capable of delivering significant benefits in the security, reliability and efficiency of our digital infrastructure at all scales.
We motivate this vision by presenting two challenges arising in different fields yet with fundamental commonalities best addressed by a unifying software platform supporting devices ranging from virtual servers in the cloud, through desktops, to mobile smartphones. After drawing out this common ground, we describe our solution and its benefits. We then describe the initial steps we have taken toward our solution, the Mirage framework, as well as ongoing future work.

1. INTRODUCTION

The Internet has experienced huge growth in the last decade, and hundreds of millions of users now depend on it for daily news, entertainment and commerce. Thousands of large datacenters have sprung up to fuel the demand for compute power to drive these Internet sites, fundamentally changing the economics of hosting in recent years. Cloud computing (§1.1) means that software infrastructure now runs on virtual machines, and usage is charged "by the second" as resources are consumed.

Simultaneously, we have seen mobile devices and social networking become popular with consumers as a way of operating on the move. Devices such as the iPhone boast fully-featured software stacks and gigabytes of solid-state storage. However, 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.

The common theme here is that software efficiency is increasingly valuable alongside traditional themes such as performance or security. However, the evolutionary nature of the computer industry has resulted in many software layers building up that are not well-suited to small, efficient outputs.

Within Horizon, our goal is to construct novel infrastructure in support of the digital economy, suitable for decades of use by society. To begin, we are building a clean-slate programming model for an energy-constrained, highly mobile society that generates vast amounts of digital data for sharing and archiving.[1] We refer to this model as *multiscale computing*, its key characteristic being that it enables the generation of binaries suitable for execution on resources ranging from individual smart-phones to sets of cloud resources.

We first briefly introduce cloud computing (§1.1) and mobile computing (§1.2) to a non-expert audience, and then motivate our vision with two different challenges: scientific computing (§2) and social networking (§3). We then draw out the commonalities between these seemingly diverse topics (§4), and describe how we address them in Mirage (§5). Finally, we review related work (§6), and discuss the current status of the project and plans for future work (§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

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

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

a single provider. Sites such as Google and Amazon started building datacenters spanning the USA and Europe, causing their reliability and energy demands also to grow.

In 2006 the EPA estimated that datacenters consumed 1.5% of the total electricity bill of the entire USA—around $4.5bn [12]—and there was a rush for hydro-electric power in more remote locations in the US. Companies were forced to over-provision to deal with peak demand (e.g., Christmas time), and thus had a lot of 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 [14].

The Xen hypervisor was developed as the low-level mechanism to divide up a physical computer into multiple chunks [2]. It grew to become the leading open-source virtualization solution, and was adopted by Amazon as part of its "Elastic Computing" service which rents spare capacity to anyone willing to pay for it.[2] Amazon became the first commercial provider of what is now dubbed "cloud computing"—the ability to rent a slice of a huge datacenter to provide on-demand computation resources that can be dynamically scaled up and down according to demand.

Cloud computing has proved popular in the era of Web 2.0 services, which experience frequent "flash traffic" that require large amounts of resources for short periods of time. Other uses are for one-off tasks, such as the New York Times scanning their back catalogue of 11 million articles from 1851 to 1922 into PDF format for on-line delivery. The whole process took 100 virtual machines over a 24 hour period [16]. Finally, companies with large data processing requirements have been constructing distributed systems to scale up to thousands of machines, e.g., Google's BigTable [6] and Amazon's Dynamo [9].

The evolutionary nature of the rise of cloud computing has led to some significant economic and environmental challenges. Virtualization encapsulates an entire operating system and emulates 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 [33].

In the context of a large datacenter, these layers are an efficiency disaster: they potentially add millions of lines of code performing redundant operations. It is here that we position our work: compressing all these layers into just one that executes 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 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 on their devices.[3]

We observe a similar shift toward distributed computation in the mobile space, with use of crowd-sourcing to collect sensor data from a global audience. Each individual mobile phone is quite powerful and so, with only limited co-operation, can enable impressive distributed computation such as the assembly of 3D models of entire cities [1].

Devices such as the iPhone and Android phones run commodity operating systems (OSs)—OSX and Linux respectively—with selected customisations. For example, both those platforms remove page swapping

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

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

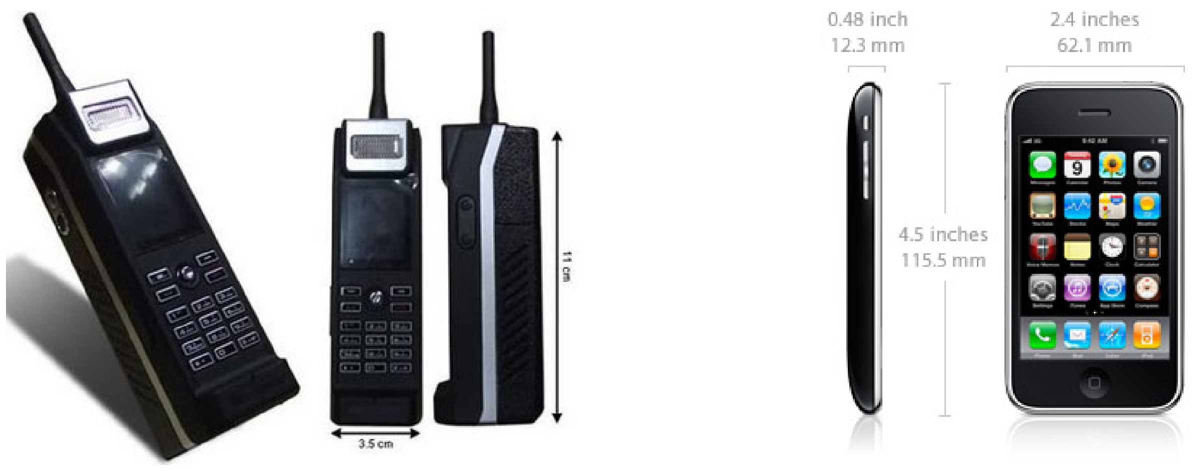

Figure 2: *Comparing a phone from a couple of decades ago with a modern Apple iPhone*

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 of those platforms view swapping out individual pages as of little benefit.

Notwithstanding such customisations, *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.

2. SCIENTIFIC COMPUTING

Over the last 20–30 years, many branches of science have undergone a revolution in terms of the sizes of datasets that are being gathered and processed: datasets are now massive, containing millions, billions and more datapoints. For example, gene sequencing generates hundreds of gigabytes of data with a single experimental run, and many terabytes or even petabytes over the course of an experiment. Modern astronomical instruments generate multi-MB images containing millions of pixels, e.g., HERSCHEL generates 4 MB images from a 1024×1024 CCD camera, and each will take many thousand such images over its lifetime. In particle physics, instruments such as the Large Hadron Collider at CERN are expected to generate data at 1.5 GB/s for over a decade when fully operational [29].

Datasets need not be observational in nature and may present challenges other than sheer data volume: the Millenium Run, a large-scale cosmological simulation, involved simulating 10 billion "particles" each representing about a billion solar masses of dark matter, in a cube region about 2 billion light years in length [31]. The simulation used a super-computer in Garching, Germany for over a month and generated 25 TB of output data.

A consequence of the availability of these massive datasets is that scientific and statistical methods are evolving: it is no longer necessary to extract all possible information from tens or hundreds of datapoints. Instead, the problem is how to build data processing pipelines that can handle these enormous datasets, reducing them to manageable pieces of information, often for subsequent, more traditional, analysis. When dealing with such large datasets, cloud computing is often the only economically feasible approach: since these massive computation farms of hundreds or thousands of machines are only sporadically required for a few days at a time, it is just not worth purchasing, operating and maintaining them for the entire lifetime of a project basis. Instead, cloud facilities such as Amazon AWS are used as-needed.

At the same time, the software for these processing pipelines needs to be developed, using manageable but representative fragments of the full dataset. Code development is generally best carried out using the scientist/developer's local compute platform, typically a laptop or desktop PC. Development like this is greatly accelerated if the local runtime environment is closely matched to the cloud environment in which the code will be run over the entire dataset: a prime example of multiscale computing.

3. PERSONAL CONTAINERS

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.[4] The largest site, Facebook, has over 300 million registered users, all sharing photos, movies and notes. Personal privacy has suffered due to the rapid growth of these sites [21] 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.[5]

This dependence on commercial providers to safeguard data also has significant wider societal implications. The British Library warned "historians face a black hole of lost material unless urgent action is taken to preserve websites and other digital records."[6] 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 that 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" 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 benefits from a new multiscale 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 systems principles as email, in which no central system has overall control. Instead, individuals direct queries to their personal container, whether it is running on their phone, at their home, or in the cloud. In this way we liberate our data from the control of a few large corporations, reclaiming it to ourselves. 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. Furthermore, personal containers are envisaged running on such a wide range of platforms that the software stack in question must be multiscale.

4. MULTISCALE NOT MULTICORE

In the previous sections we presented quite different applications which nonetheless share common ground: the need for multiscale computing. We distinguish *multiscale* from *multicore*: the benefit is gained not by being able to utilise the many cores now available on modern processors, but by being able to scale the application up to make use of available resources, whether one or many handheld devices, or a virtual datacenter in the cloud.

[4] `http://www.nielsen-online.com/pr/pr_090713.pdf`

[5] 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`

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

Note that we are not addressing the general problem of how best to segment a particular problem for parallel execution: it is still up to the programmer to manage distribution of work across multiple available resources. However, our approach should make the programmer's life easier by providing a consistent programming framework to help them manage their use of the available resources, as well as modern tools such as type-checking to help them manage common problems such as concurrency.

The basic underlying principle of multiscale computing is that of simplicity, giving rise to attendant benefits such as efficiency, reliability, and security. Current software stacks, such as LAMP,[7] are too "thick," containing now unnecessary support for legacy systems and code. By using results from the last 20 years of computer science research in fields such as privacy, virtualization, languages (both practical runtimes and theoretical underpinnings), we 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 cloud-hosted virtual machines. This allows the programmer to focus on their program without needing to explicitly manage differences in storage, networking and other infrastructure.
- *Efficiency*. As noted earlier, the legacy support in modern software stacks introduces overheads, wasting energy and as a result, money.[8] Further, services such as AWS are now introducing "spot pricing," providing the ability for consumers to bid for spare capacity, paying in accordance with demand.[9] 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 providing a single bootable image, generated from code written in a strongly typed and automatically type-checked language, the reliability of programs is increased. Additionally, since the virtual machine's configuration is compiled into the image, it becomes possible to reason about policies applied when accessing resources such as the network, further increasing the application's security.

Although we rely heavily on recent research in the areas mentioned, we believe there remain further research challenges in fully realising this vision; we briefly note some of them here:

- *Programming languages*. How can we ensure that personal containers are secure and efficient enough to hold a lifetime of personal data?
- *Ubiquitous computing*. How do we enable personal containers to be run on diverse hardware platforms such as mobile devices or online cloud computers?
- *Resource management*. How do we efficiently handle storage, encryption and history, and ensure easily availability without losing privacy?
- *Formal methods*. Can we mathematically reason about both our stored data itself and our access to it, to make stronger privacy guarantees while still allowing useful datamining?

In the following sections we describe the initial steps we are taking to address these challenges with the *Mirage* platform in the context of the Horizon Institute.

5. THE MIRAGE APPROACH

Mirage is a clean-slate system that compiles applications directly into a high-performance, energy-efficient kernel that can run directly on mobile devices and virtual clouds. Applications constructed through Mirage become operating systems in their own right, able to be targeted at both Xen-based cloud infrastructure, and ARM-based mobile devices. Mirage can also output binaries which execute under POSIX-compliant operating systems, aiding development and debugging. Mirage also exposes new capabilities directly to the software, allowing applications to make more effective use of live relocation [7], resource hotplug [38], and low-energy computation modes.

In this section, we first describe how we simplify the software stack by eliminating runtime abstractions (§5.1). We also take advantage of advances in modern programming languages to perform a number of checks at compilation time, so they can be removed from running code (§5.2). Continuing in the same spirit, Mirage *statically specializes* applications at compilation-time based on configuration variables. This eliminates unused features entirely, keeping the deployed executables smaller (§5.3). Finally, we provide brief overviews of the concurrency model (§5.4) and storage (§5.5) in Mirage.

5.1. Reducing Layers

Although virtualization 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

[7]Linux, Apache, MySQL, PHP
[8]We leave it to the reader to decide which is more important.
[9]`http://cloudexchange.org/`

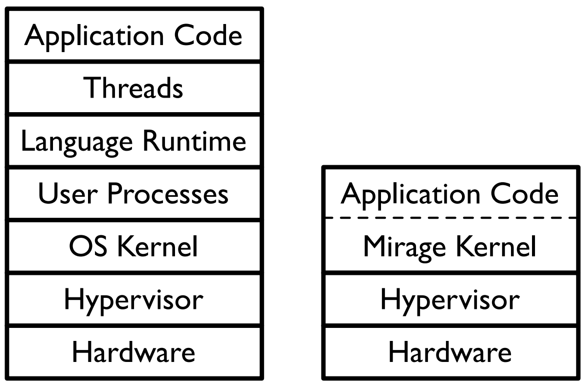

Figure 3: *A conventional software stack* (left) *and the Mirage approach with statically-linked kernel and application* (right)

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).

The combination of operating system kernels (comprising millions of lines of code) and language runtimes (also typically millions of lines of code) has lacked synergy due to the strong divide between "kernel" and "userspace" programming. Standards designed decades ago, such as POSIX, have dictated the interface to the kernel, and language runtimes have been forced to use this inefficient legacy interface to access physical resources even as hardware has 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 (§5.4).

A typical functional language 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 a UNIX-like operating system, 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 some harmful interactions—consider a server which invokes *fork(2)* for every incoming network connection. The garbage collector, now present in multiple independent processes due to the fork, will sweep across memory to find unused objects, reclaim them, and compact the remaining memory. This 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 if 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 functional language implementations. Others include the process abstraction, as the static type safety guarantees made by typical functional languages eliminate the requirement for hardware memory protection unless there remain other unsafe components in the system. As Mirage is intended for applications written exclusively in type-safe languages, we can eliminate a number of these layers of isolation, in turn reducing the work done by the system at runtime.

The inefficiency of functional languages compared to C or Fortran has traditionally been cited as the reason they are not more widely adopted for tasks such as scientific computing, along with the lack of mature tool-chains and good documentation [36]. We are aiming to close the performance barrier to C with Mirage, as with our previous work in this area [26; 24].

5.2. Objective Caml

Mirage is written in the spirit of vertical operating systems such as Nemesis [17] or Exokernel [10], 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). These features mean that it provides a stronger basis for the practical application of formal methods such as model checking; and the removal of redundant safety system checks greatly improves the energy efficiency of the system.

Objective Caml [23] (or OCaml) is a modern functional language based on the ML type system [22]. 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 [20] exist which convert precise formal specifications of algorithms into certified executables.

We do not modify the OCaml compiler itself, but rather the runtime libraries it provides to interface OCaml with UNIX or Win32. This code is mostly written in C, and includes the garbage collector and memory allocation routines. We recreate from scratch the built-in library to support a subset of the standard features (e.g., we omit threading and signal handling). The application is structured as a single co-operative thread and executed until that thread terminates.

5.3. 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 [30].

Specialized devices such as network routers split these concerns into: (*i*) a general-purpose control plane; and (*ii*) a specialized 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.

We treat a general purpose OS such as Linux as the Control Operating System (COS). The COS, running under Xen as a guest, exists purely to configure, compile, and launch a Mirage application. It does not provide application services to the outside world, and is accessed purely through a management interface.

An example of specialization for a web server would be to compile read-only static content into the binary, along with a database library to handle more dynamic queries. If write interfaces aren't required, then they are simply not compiled in to the final binary, further reducing its size.

Static specialization simplifies the memory layout of a Mirage instance, since it results in a single executable kernel which is directly executed by Xen as a guest operating system. At start-of-day, Xen allocates a fixed number of memory pages to Mirage, all in a single address space; since OCaml is type-safe, we do not require multiple virtual address spaces. The general layout for 32-bit x86 processors is illustrated in Figure 4 and has the following regions: (*i*) the top 64MB reserved for the Xen hypervisor; (*ii*) the code area of the executable program's instructions and static data; (*iii*) the network area where pages for remote data traffic are mapped; and (*iv*) the heap area managed by the garbage collector.

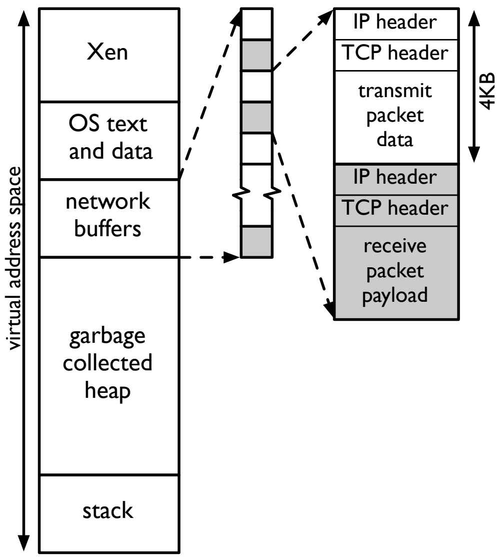

Figure 4: *Virtual memory layout of a Mirage kernel*

The exact sizes allocated to each of the regions are configurable, but can be aligned to megabyte boundaries to facilitate the use of large page sizes, gaining the benefits of reduced TLB pressure.

The runtime libraries (e.g., the garbage collector and interface routines to Xen) are written in C, and are thus potentially unsafe. To simplify the code and make static analysis easier, the runtime is single-threaded and dynamic memory allocation is minimized. The standard OCaml garbage collector is used to manage the dynamic heap area, with memory allocation routines being modified to the new address layout. Excess heap and stack pages are loaned back to the hypervisor when not in active use.

5.4. 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 [34]. Our I/O threads are based on the Lwt co-operative threading library [35] which has a syntax extension to hide the yield points from the programmer, giving the illusion of normal pre-emptive threading.

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.[10]

A useful feature of Lwt is that the blocking nature of a function is exposed in its OCaml type, letting the programmer be certain when a given library call might take some time. Any blocking function can be macro-expanded at compile-time to block more often, for example with a high-level debugger. We have previously explored these ideas while integrating model-checking into server applications [25] and are now prototyping a web-based debugger specially designed to find problems in Mirage code.

For running computations that require more than one core, we adopt a parallel task model based on the join calculus [13], 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 task 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 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 exploit the cross-platform aspect. 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 to perform the actual processing. 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.

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

5.5. Storage

These days data storage is cheap and we have to deal with massive datasets. Scientists, market analysts, traders and even supermarkets compute statistics over huge datasets often gathered globally. We have already recounted examples of the size of modern scientific computing experiments (§2).

Multiscale computing thus needs low-overhead access to large datasets, as well as the ability to quickly partition it for distribution across smaller devices such as desktop computers or mobile devices. Since Mirage is designed to be platform-independent, it does not expose a conventional filesystem, but instead has a persistence mechanism that operates directly as an extension to OCaml. We use multi-stage programming [32] to analyze user-defined types at compilation time and automatically generate the appropriate code for the target hardware platform.

This approach has several advantages: (*i*) the programmer can persist arbitrary OCaml types with no explicit conversion to-and-from the storage backend; (*ii*) code is statically generated that can work efficiently on the target platform (e.g., SQL on phones, or direct block access in the cloud); and (*iii*) a high degree of static type-safety is obtained by auto-generating interfaces from a single type specification.

The storage backend is immutable by design, and so changes to data structures cause records to be copied. This means that it is naturally versioned and reverting to older versions is easy. Values no longer required can be released and, as with the memory heap, a storage garbage collector cleans up unused values. The use of an integrated and purely functional storage backend is a key design decision for Mirage as it permits many parallel workers to “fork” the storage safely and cheaply, even if a huge dataset is being shared. Concurrent access to records is guaranteed to be safe due to the immutability of the backend.

This immutability overcomes one of the biggest barriers to using the online cloud for large datasets—getting the data in and out is expensive and slow.[11] An immutable data store ensures that a single copy of the data can be

[11] To mitigate this, Amazon offers physical import/export to EC2, see `http://aws.amazon.com/importexport`

worked on by many consumers, and avoiding the need to fork even if the operations being performed on it are very different.

In scientific computing most datasets are written as streams and rarely modified. Instead, analysis code is run over the data and derived values computed, which are often hard to track as they depend on the versions of both the input dataset and the code used. We are prototyping a version-controlled database which tracks both values *and* code side-by-side, ensuring that scientists can easily trace the provenance of potentially huge datasets by examining the code that generated the intermediate analyses. For personal containers, the database lets us easily archive personal history as it changes. Older data can be "garbage collected" into cheaper, off-line archival storage while current data remains live and searchable from the database.

This approach is not initially intended to be suitable for all purposes (e.g., it would struggle with mutation-heavy richly-linked graph data), but we anticipate that it will find key uses in dealing with read-heavy data that can be stored a single time in the cloud and made available to multiple users safely and efficiently. For development, the storage extension also has a SQL backend to integrate with existing database servers and work with smaller slices of the larger datasets [15].

6. RELATED WORK

The Mirage operating system is structured as a "vertical operating system" in the style of Nemesis [17] and Exokernel [10]. 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 [28].

More recently, Barrelfish [3] 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 [8]. Mirage aims to remove the distinction between multicore and clustered computers entirely, by adopting multiscale 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 [19], 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 [26] and a lighter statically-typed runtime. Our multiscale threading model also helps target smaller mobile devices which would struggle to run the full CLR environment.

Parallel processing of large datasets is an active area of research, e.g., direct language support with Data Parallel Haskell [5] or software transactional memory [18]. Our main contribution here is to provide an efficient runtime for single-core processing, and a common immutable data store to allow large clusters of cloud-based VMs to operate over the data.

The concept of Personal Containers was inspired by the Personal Server from Intel Research [37]. The Personal Server is a small, efficient mobile device carrying personal data and wirelessly communicating with other devices. We seek to generalise this concept into containers that work seamlessly between the cloud and personal devices, and also provide a programming model to make it easier to write secure, fast, ubiquitous computing applications.

7. STATUS

We are in the process of bringing this vision into reality. We have the first revision of Mirage running,[12] targeting Xen cloud systems and desktop operating systems, and are working on retargeting for ARM-based handhelds. There are many "bare-metal" optimizations in progress for the cloud target which promise improved performance, as well as integrating improved versions of our previous work on fast, type-safe network parsing [26] and model-checking [25]. Even unoptimised, the runtime library 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.

The database backend has been implemented with a SQL backend to ensure that the syntax extension, immutable semantics and type-introspection algorithms are sound. It is available for standalone use in other OCaml code as open-source software.[13] The custom cloud backend for Xen will have the same behaviour but with higher performance due to the removal of the SQL translation layer.

Personal Containers have been prototyped in collaboration with the PRIMMA project[14] in the form of Personal Butlers which guard user's privacy online [39]. We aim to release a first public and open-source version of

[12]`http://mirage.github.com/`

[13]`http://github.com/mirage/orm`

[14]`http://primma.open.ac.uk/`

personal containers in 2010, with support for a variety of protocols such as IMAP and POP3, Facebook integration, and local application support for MacOS X.

We are now investigating possible deployment prospects within work being carried in the Horizon Institute.[15] This is looking at research under the broad heading of the Digital Economy, with infrastructure one of its three main challenges. We aim to make use of Mirage to build some of this infrastructure allowing us to deploy and test both the Mirage multiscale OS and Personal Containers at scale.

We are also collaborating with hardware vendors to investigate the possibility of pushing the approach taken by Mirage even further down the stack, e.g., into the networking hardware itself using SR-IOV and smart NICs [27], allowing even greater optimisation possiblities in terms of the efficiency and safety of ICT equipment. 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 [4] and linear typing [11]. These allow more dynamic checks to be eliminated safely at compilation-time, further improving the efficiency of runtime code without sacrificing safety properties.

[15] `http://www.horizon.ac.uk`

REFERENCES

[1] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski. Building Rome in a day. In *International Conference on Computer Vision*. IEEE, 2009.

[2] 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.

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

[4] 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.

[5] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal - nested data parallelism in haskell. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 524–534, London, UK, 2001. Springer-Verlag.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] T. Gazagnaire and A. Madhavapeddy. Statically-typed value persistence for ML. In *Proceedings of the Workshop on Generative Technologies (WGT'2010)*, March 2010.

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

[17] 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.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

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

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

[21] 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.

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

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

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

[25] A. Madhavapeddy. Combining static model checking with dynamic enforcement using the statecall policy language. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 446–465. Springer, 2009.

[26] 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.

[27] 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.

[28] 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.

[29] G. L. Presti, O. Barring, A. Earl, R. M. G. Rioja, S. Ponce, G. Taurelli, D. Waldron, and M. C. D. Santos. Castor: A distributed storage resource facility for high performance data processing at cern. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 275–280, Washington, DC, USA, 2007. IEEE Computer Society.

[30] 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.

[31] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(7042), 2005.

[32] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50, Dagstuhl Castle, Germany, March 2004. Springer.

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

[34] 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.

[35] 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.

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

[37] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The Personal Server: Changing the way we think about ubiquitous computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 194–209, London, UK, 2002. Springer-Verlag.

[38] 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.

[39] R. Wishart, D. Corapi, A. Madhavapeddy, and M. Sloman. Privacy Butler: A personal privacy rights manager for online presence. In *IEEE Percom Workshop on Smart Environments 2010*, September 2010.