Turning Down the LAMP: Software Specialisation for the Cloud

Anil Madhavapeddy[1], Richard Mortier[2], Ripduman Sohan[1], Thomas Gazagnaire[3]
Steven Hand[1], Tim Deegan[3], Derek McAuley[2] and Jon Crowcroft[1]
University of Cambridge[1], University of Nottingham[2], Citrix Systems R&D[3]

Abstract

The wide availability of cloud computing offers an unprecedented opportunity to rethink how we construct applications. The cloud is currently mostly used to package up existing software stacks and operating systems (e.g. LAMP) for scaling out websites. We instead view the cloud as a stable hardware platform, and present a programming framework which permits applications to be constructed to run directly on top of it without intervening software layers. Our prototype (dubbed Mirage) is unashamedly academic; it extends the Objective Caml language with storage extensions and a custom run-time to emit binaries that execute as a guest operating system under Xen. Mirage applications exhibit significant performance speedups for I/O and memory handling versus the same code running under Linux/Xen. Our results can be generalised to offer insight into improving more commonly used languages such as PHP, Python and Ruby, and we discuss lessons learnt and future directions.

1 Introduction

Cloud computing has changed the economics of hosting in recent years. As large datacenters have grown to provide modern Internet services, deployment of virtualisation platforms has enabled their computing power to be rented by customers to run their own applications. The first such service was Amazon's *Elastic Computing* [1], which provides resources to customers dynamically and scales up and down according to demand.

Unfortunately, this extremely dynamic resource availability within the cloud is not well supported by traditional software stacks, e.g., LAMP (Linux, Apache, MySQL, PHP). These are very "thick," containing extensive support for legacy systems and code built up over years. This makes them cumbersome to build and deploy, inefficient to run, and complex to administer securely. Some of these concerns are being tackled via prepackaged binary images, bolt-on "accelerators" which dynamically optimise scripting code such as PHP, and automated security update infrastructure.

In this paper we propose departing from this approach of layering systems, instead developing a software stack designed explicitly for use in the cloud. Standard interfaces such as POSIX are less relevant in this highly distributed environment, and a fresh software stack can also help to exploit the *new* capabilities of virtualisation more effectively, such as live relocation. Software efficiency now brings direct financial rewards in cloud environments, providing a much greater impetus to improve on the current state-of-the-art and reducing resistance to change from the open source community (e.g. the "NoSQL" movement [15]).

This paper makes several contributions:

(*i*) the motivation for constructing a new software stack (§2), and its architecture (§3);

(*ii*) early performance results that demonstrate significant improvements in I/O and memory speeds, suggesting this is an area worth exploring (§4); and

(*iii*) a discussion which generalises our results to other programming frameworks (§5).

Finally, we examine related work and conclude (§6).

2 A New Approach: Mirage

The key principle behind Mirage is to treat cloud virtual hardware as a compiler target, and convert high-level language source code directly into kernels that run on it. Our prototype compiler uses the OCaml language (§3) to further remove dynamic typing overheads and introduce more safety at compile time. We now break down our design considerations into efficiency, security, simplicity, purpose, and ease-of-use, as follows:

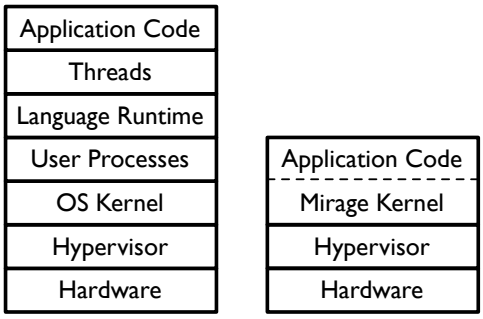

Figure 1: A conventional software stack (*left*) and the statically-linked Mirage approach (*right*).

Efficiency Multiplexed interfaces in modern software stacks generate substantial overheads which, when deployed at scale, waste energy and thus money [32]. Making efficient use of the cloud requires the ability to instantiate and destroy virtual machines (VMs) with low latency, difficult with today's heavyweight software stacks such as LAMP. We discuss concurrency (§3.2) and storage (§3.3) later, and how Mirage improves the situation.

Security Mirage applications are bootable images generated from code written in a strongly type-checked language. The language eliminates low-level memory issues such as buffer overflows. The application's configuration is analysed during compilation to remove unused features, reducing the attack surface for attackers while also reducing memory usage.

Simplicity The semantics of many OS interfaces are inconsistent, due to having to support a wide application base which has evolved over time—both networking [4] and filesystem [23] APIs have known deficiencies. OS kernels often favour performance over strict safety [6], and when applications such as databases need strong guarantees (e.g., writing blocks to disk), they are forced to make conservative assumptions and suffer performance drops.

Current software stacks are 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. The main driver for virtualisation to date has been to consolidate under-utilised physical hosts, and this adds another runtime software layer with the hypervisor. This is essential to run existing OSs and software to run unmodified (Figure 1 *(left)*).

Focus Most operating systems try to solve everything—from desktops to network servers to fast-paced games. In some cases, such as concurrency, researchers have spent a lot of time examining individual models, e.g., threading versus event architectures [34, 37], and a general-purpose kernel has to support all of them. Mirage focusses on the domain of I/O intensive cloud servers, which lets us specialise the stack (see Figure 1 *(right)*) and reap the benefits.

Ease of Deployment There have been many brave attempts in the past to build systems like this which use high-level languages [11, 13] or radically different designs [8], but they all hit the issue of research OSs becoming rapidly obsolete with advances in physical hardware [24]. In contrast, the hypervisors being used for cloud computing (mostly Xen and Hyper-V) provide a standard but complex and low-level interface to program against [26]. Mirage hides this complexity behind the compiler toolchain, and gives a programmer usable, high-level language abstractions whilst targeting the low-level virtual hardware interface directly.

3 Design and Implementation

Objective Caml [17], or *OCaml*, is a modern functional language supporting a variety of programming styles, including functional, imperative, and object-oriented. It is a dialect of the ML family, with a well-designed, theoretically-sound type system that has been developed since the 1970s [16].

ML is a pragmatic system that strikes a balance between 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 references and mutable data structures while *guaranteeing* that all such side-effects are *always* type-safe and will never cause memory corruption. Safety is achieved by two methods: (*i*) static type-checking; and (*ii*) dynamic bounds checking of array and buffers.

We have previously shown how to construct secure, high-performance network applications using OCaml [19]. Our applications were largely OS-independent, and so a simpler runtime dedicated to the purpose of running them could deliver significant cost savings and performance improvements in cloud environments—now the central goal of Mirage. We do not modify the OCaml compiler itself, but rather the runtime libraries it provides to interface with the OS. This code is mostly written in C, and includes the garbage collector and memory allocator.

3.1 Memory Management

The static type safety guarantees made by OCaml eliminate the need for runtime hardware memory protection, except against other unsafe system components. Mirage runs applications in a single 64-bit virtual address space (see Figure 2). Modern OS kernels discourage static

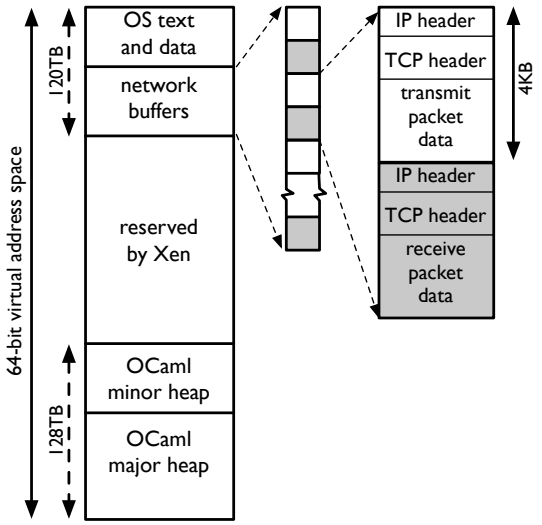

Figure 2: Virtual memory layout of a 64-bit Mirage kernel running under Xen.

memory mapping in favour of Address Space Randomisation (ASR) for protection against buffer overflows [28]. Mirage does not benefit from ASR since the application is type-safe, and the runtime itself has little dynamic allocation and no shared library mappings.

In 64-bit mode, the bottom 128TB and the top 120TB of virtual address space are available for use by the guest, with the rest either reserved by the hypervisor or non-canonical addresses. The application begins with code and static data mapped into the bottom portion of virtual memory, and the OCaml heap area and network buffers placed separately into the top 120TB region. Pages are transmitted and received in Xen by *granting* them to the hypervisor [36]. The exact size allocated to each region is configurable, but is aligned to 2MB boundaries to use `x86_64` "superpages" resulting in smaller page-tables.

Statically mapping virtual memory in this fashion provides Mirage with a significant speed boost. Under normal kernels, the standard OCaml garbage collector cannot guarantee that its address space is contiguous in virtual memory and maintains a page table to track the allocated heap regions. In tight allocation loops, the page-table lookup can take around 15% of CPU time, an overhead which disappears in Mirage (see Figure 4).

3.2 Concurrency

In Mirage, we collapse all concurrency into two distinct models: (*i*) lightweight control-flow threads for managing I/O and timeouts; and (*ii*) optimised inter-VM communication for parallel computation. Each Mirage instance runs as on a single CPU core, and depends on the hypervisor to divide up a physical host into several single-core VMs. Thus, a parallel program runs as eight VMs on a single eight-core machine, or be spread across four two-core physical hosts. Communication is optimised dynamically: if the VMs are running on the same physical machine, Mirage uses shared memory channels instead of the network stack.

We believe that these two models are sufficient to capture the main uses of parallel programming without running into the difficulty of creating a unified model suitable for both large-scale distributed computation and highly scalable single-server I/O [33]. Our control threads are based on the Lwt co-operative threading library [35], and has syntax extensions to permit programming in a similar style to pre-emptive threading. Although we do not present a full evaluation of threading in this paper due to space limitations, informal benchmarks are available for Lwt which illustrate our points [9].

3.3 Storage

Mirage provides a persistence mechanism as a non-intrusive language extension to OCaml [12]. For each datatype specified by the programmer, we statically generate functions at compile time to save and retrieve values of this type to and from the Xen virtual block devices. This approach has several advantages: (*i*) the programmer can persist arbitrary OCaml types with no explicit conversion: the details are completely abstracted away; (*ii*) the code is statically generated and highly optimised for efficiency; and (*iii*) better static safety is guaranteed by auto-generating interfaces. For example:

```
type t = { name: string; mail: string } with orm
let me = { name="Anil"; mail="avsm2@cam.ac.uk" }
let main () =
 let db = t_open "contacts" in
 t_save db t;
 let cam = t_get ~mail:('Contains "ac.uk") db in
 printf "Found %d @ac.uk" (List.length cam)
```

The type `t` is a standard OCaml type, with an annotation to mark it as a storage type. Variables of type `t` can be saved and queried via the `t_open`, `t_save` and `t_get` functions.

The backend uses the SQLite database library, and SQL is automatically generated from the application's datatypes and never used by the programmer directly. SQLite has a VFS layer to let OSs define their own I/O mechanisms. Mirage implements a Xen `blkfront` VFS which interacts directly with a block device without an intervening filesystem. This layer performs highly optimised I/O: (*i*) all database read/writes are 4KB page-aligned and contiguous segments use `blkfront` scatter-gather allowing up to 11 pages to be sent in one request; (*ii*) the journal file is mostly appended to, and only needs to be flushed when the VFS layer requests

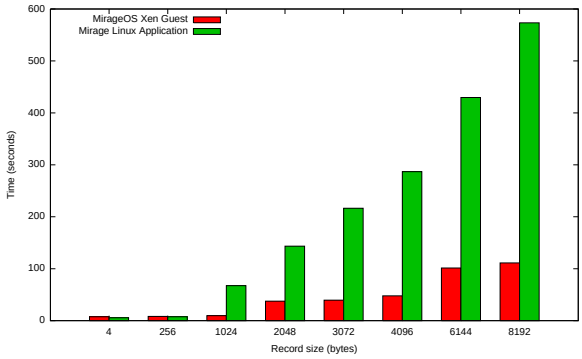

Figure 3: Performance of a SQL stress test running as a direct kernel *vs.* 64-bit Linux userspace.

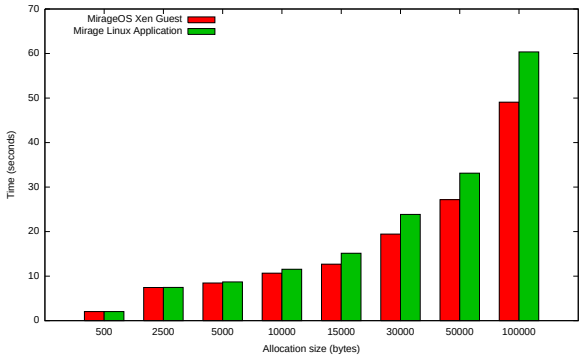

Figure 4: Allocation performance under Mirage *vs.* 64-bit Linux, for 100 million varied-size allocations.

a sync; (*iii*) write barriers can maintain safe journal semantics without needing to wait for a synchronous disk flush. Although some of these assumptions are invalid for general-use filesystems, they work very well with a database engine.

4 Evaluation

So far, we have asserted several performance benefits to running Mirage kernels. We now confirm our hypothesis via performance tests on Mirage as kernels versus on virtualised Linux. Tests were conducted on a quad-core Intel Xeon 2.83GHz system with 4GB RAM, running Xen 3.4.3 and a 64-bit Linux 2.6.27.29 dom0 kernel. The guests were configured with 1GB RAM, 1 vCPU, and LVM block storage. The Linux guest used the same 64-bit PV kernel as dom0 and an `ext2` filesystem.

We evaluated the performance of our database storage by running a benchmark that inserts, updates and deletes records of varying sizes over 5000 iterations. The database was checked after each iteration and all values compared to ensure integrity.

Figure 3 shows the results of this benchmark with varying record sizes. Linux is faster for small record sizes but performance decreases as individual record sizes increase. In contrast, Mirage performance is significantly faster and scales better with increasing record size. We attribute this to the specialised buffer cache in Mirage which takes advantage of page-aligned direct I/O, as well as the optimised heuristics for journal files (which are typically short-lived and only require flushing to disk once, memory permitting). One of the main benefits of implementing the SQLite VFS operations in OCaml is to let us experiment with different heuristics and data structures more easily; one of the main benefits of Mirage is exactly that it makes this kind of specialisation straightforward.

We tested memory allocation performance by allocating 100 million strings of varying sizes (see Figure 4). Again, Mirage is faster than Linux for larger allocations, illustrating the benefits of running without a kernel/userspace divide. x86_64 does not have segmentation, and Xen protects its own memory using page-level checks and runs both the guest kernel and userspace in ring 3 [25]. This makes system calls and page table manipulation relatively slow, a problem which Mirage avoids by not context-switching in ring 3.

In terms of binary size, the Mirage images for the benchmarks were around 600KB in size, in contrast to Linux distributions which are difficult to squeeze below 16MB, or Windows which runs into hundreds of megabytes.

5 Lessons Learnt and Future Directions

Although it is a great platform for trying out ideas, OCaml is dwarfed by the vast user-bases of Ruby, Python and PHP. We now consider some of the lessons that can be pulled out to improve other frameworks, and our plans for interesting directions based on them:

Multi-scale not multi-core There is much concern about multi-core scalability of scripting languages (which often have global interpreter locks). In a virtual platform, this simply isn't as important as scaling widely across hosts and letting the hypervisor divide up cores. Frameworks which currently use (for example) *fork(2)* on a host to spawn processes would benefit from using cloud management APIs to request resources and eliminate the distinction between cores and hosts. Indeed, the lack of a standard interface from within a guest to cloud APIs (as an analogue to kernel syscalls from userspace) is a glaring omission. We are currently building a system of cloud syscalls with a view to constructing *self-scaling applications* that can alter their resource

usages dynamically in a cloud environment. For large-scale computation, our Skywriting system [22] provides a task-parallel coordination language that provides efficient job dispatch for individual hosts and their cores, leaving Mirage instances to do the actual computation work on reasonably sized chunks of data.

Type-driven meta-programming Using code to generate more code (a key aspect of our storage mechanism) is also used in dynamic languages to interface to databases (e.g. ActiveRecord in Ruby on Rails). However, this generated code does not need to be the *same* language as the host, and so dynamic languages could derive a lot of free performance by using code-generation (e.g. LLVM [14]) to approach the performance of our statically typed implementation.

One of the more exciting examples of this in Mirage is the ability to run across multiple cloud provider's database systems. An application can run on Amazon EC2 using virtual block devices, and the *same* source code also works directly against Google AppEngine using the Datastore API (via an OCaml-to-Java compiler [5]). Support for this only required adding a code generator to our storage compiler, and we are also implementing Amazon SimpleDB and Tokyo Cabinet backends. This research direction offers a solution to one of the biggest problems with using cloud infrastructure—vendor lock-in—since the same source code could run on several vendor's infrastructure such as Amazon and Google (both of whom have had high-profile outages in 2009).

Fat-free code Code size and startup time has traditionally been poor for scripting languages. It is of crucial importance for client-side Javascript frameworks like jQuery that offer highly compressed "min" versions, and normal versions for development use. In a cloud environment, traffic spikes (e.g. a link from a popular media site) can drive load up by orders of magnitude in seconds. A framework which can start instances very quickly (boot kernel, load code, prime caches, establish database connections) is well positioned to minimise the use and cost of cloud resources. Mirage demonstrates how little code is actually required from a cloud kernel, with a typical web-server binary around 400Kb (and with further room for improvement). Facebook has taken initial steps in this direction by pre-compiling PHP to C++/Linux binaries using its HipHop compiler [39], and we are looking into porting this to run directly under Mirage.

Dont forget to be virtual Cloud platforms offer several features not easily available on physical hosts, such as live relocation. Currently, this happens at the OS level, and applications have little control over the process. The simple memory layout in Mirage (§3.1) allows us to optimise live-relocation performance (by garbage collecting before it begins, and only allocating to the minor heap), and also to propagate such events to the application via callbacks. This would permit, e.g. a web server node to update a DNS entry when it migrates, or a game server to send more frequent updates to clients to account for network disruption.

Push the limits of packaging There are a number of packaging systems which assemble small, crunched distributions suitable for cloud deployment [38], but odd limits still exist in the vendor infrastructures. For example, the biggest cloud computing vendor—Amazon—does not permit users to upload their own kernels, instead limiting them to user-space images or providing custom kernel modules. This means that a Mirage application must bootstrap itself as a Linux kernel module (overwriting the Amazon-supplied kernel once it has booted) in order to run in this environment. The reasons for this are probably not technical, but to do with supportability of custom compiled kernels or new operating systems, but it will impact the deployment of vertical operating systems like Mirage somewhat.

6 Related Work and Conclusions

Mirage draws on several research projects for its inspiration. It is structured as a "vertical operating system" in the style of Exokernel [8], and provided as a series of component libraries like Flux OSKit [10].

Barrelfish [2] provides an environment specifically for multicore systems, using fast shared-memory communication between processes. Mirage uses a hypervisor as its scheduler, focusing on fast single-vCPU execution. The Extremely Reliable OS (EROS) had a similar focus on removing runtime layers [30], and its single-level store KeyKOS [29] extended the memory system all the way to the disk, as our language-integrated storage also does.

We derive a great deal of inspiration from Foxnet [3], which built a TCP/IP stack in Standard ML. However, they reported very slow performance, and we aim to capture their elegant interface while meeting modern performance demands. Functional programming has enjoyed more industrial attention in recent years, in mission-critical systems [7], the financial sector [21], the systems community (the Xen control stack is written in OCaml [27]) and even the latest Microsoft language [31]. We believe that the cost-saving opportunity with Mirage in the cloud will be a great motivating factor encouraging the adoption of statically-typed languages.

In conclusion, Mirage is a new approach to specialising software for cloud computing environments to improve efficiency and thus cut costs. Mirage rethinks several aspects of application design by integrating networking and storage closely with the hardware and language being used. A prime design goal is *simplicity*, making formal analysis and debugging easier.

We are concentrating on release a full implementation at this stage, with several portions already available as open-source [18]. We are using it to construct large-scale services dedicated for the cloud, such as version-controlled databases for scientific computing, and "personal containers" which provision VMs where individuals can store their personal data in privacy [20]. We thank Jake Donham, Marius Eriksen, Ian McEwan, Stephen Kell, Derek Murray and Steven Smith for helpful feedback and debate, and Amazon Web Services for an AWS in Education Research Grant.

References

[1] Amazon Web Services. `http://aws.amazon.com`.

[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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44, New York, NY, USA, 2009. ACM.

[3] E. Biagioni. A structured TCP in Standard ML. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, pages 36–45, New York, NY, USA, 1994. ACM Press.

[4] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 265–276. ACM Press, 2005.

[5] X. Clerc. Cadmium. `http://cadmium.x9c.fr`.

[6] J. Corbet. Barriers and journalling filesystems. `http://lwn.net/Articles/283161`, May 2008.

[7] D. Coutts. Birth of the industrial haskell group. In *CUFP '09: Proceedings of the 2009 Video Workshop on Commercial Users of Functional Programming*, page 1, New York, NY, USA, 2009. 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*, pages 251–266, Colorado, December 1995.

[9] M. Fernandez. Comparing lightweight threads. `http://eigenclass.org/hiki/lightweight-threads-with-lwt`, 2008.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 38–51, New York, NY, USA, 1997. ACM Press.

[11] G. Fu. Design and implementation of an operating system in Standard ML. Master's thesis, University of Hawaii, 1999.

[12] T. Gazagnaire and A. Madhavapeddy. Statically-typed value persistence for ML. In *Workshop on Generative Technologies (WGT)*. ACM, Mar. 2010.

[13] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in haskell. *SIGPLAN Not.*, 40(9):116–128, 2005.

[14] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[15] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

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

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

[18] A. Madhavapeddy. Mirage. `http://github.com/mirage`.

[19] 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.

[20] A. Madhavapeddy, R. Mortier, J. Crowcroft, and S. Hand. Multiscale not multicore: Efficient heterogeneous cloud computing. In *Proc. ACM-BCS Visions of Computer Science*, Electronic Workshops in Computing, Edinburgh, UK, Apr. 2010. BCS.

[21] Y. Minsky and S. Weeks. Caml trading – experiences with functional programming on wall street. *J. Funct. Program.*, 18(4):553–564, 2008.

[22] D. Murray and S. Hand. Scripting the cloud with Skywriting. In *Hot Topics in Cloud Computing (HotCloud 2010)*, Boston, MA, USA, June 2010. USENIX Association.

[23] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), 2008.

[24] R. Pike. Systems software research is irrelevant, 2000.

[25] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.

[26] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and virtue. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[27] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions. Technical Report to appear, Citrix Systems, April 2010.

[28] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, New York, NY, USA, 2004. ACM Press.

[29] J. S. Shapiro and J. Adams. Design evolution of the eros single-level store. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 59–72, Berkeley, CA, USA, 2002. USENIX Association.

[30] J. S. Shapiro and N. Hardy. Eros: A principle-driven operating system from the ground up. *IEEE Software*, 19(1):26–33, 2002.

[31] D. Syme, A. Granicz, and A. Cisternino. *Expert F# (Expert's Voice in .Net)*.

[32] D. L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, November 1989.

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

[34] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM.

[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] 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.

[37] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[38] M. S. Wilson. Constructing and managing appliances for cloud deployments from repositories of reusable components. In *Hot Topics in Cloud Computing (HotCloud 2009)*. USENIX Association, June 2009.

[39] H. Zhao, I. Proctor, and M. Yang. Hiphop for PHP. `http://developers.facebook.com/blog/post/358`.