Using Functional Programming within an Industrial Product Group: Perspectives and Perceptions

David Scott   Richard Sharp

Citrix Systems UK R&D
Building 101
Cambridge Science Park
Cambridge CB4 0FY, UK
first.last@eu.citrix.com

Thomas Gazagnaire

INRIA Sophia Antipolis
2004 route des Lucioles
F-06902 Sophia Antipolis
Cedex, France
first.last@inria.fr

Anil Madhavapeddy

Computer Laboratory
University of Cambridge
William Gates Building
Cambridge CB3 0FD, UK
first.last@cl.cam.ac.uk

Abstract

We present a case-study of using OCaml within a large product development project, focussing on both the technical and non-technical issues that arose as a result. We draw comparisons between the OCaml team and the other teams that worked on the project, providing comparative data on hiring patterns and cross-team code contribution.

General Terms   Human Factors, Languages, Management

Categories and Subject Descriptors   D.2.m [*Software Engineering*]: [Miscellaneous]
   ;   D.3.2 [*Programming Languages*]: Language Classifications— Applicative (functional) languages

Keywords   Industry, Functional Programming, Perceptions

1.   Introduction

We present our experiences of using the programming language OCaml within the Citrix XenServer product group. The case-study is interesting for three reasons:

1. XenServer is deployed in over 40,000 companies worldwide, often in mission-critical infrastructure, with the largest single customer having more than 20,000 machines running XenServer [13]. We are presenting a very "real-world" use of functional programming.

2. It provides insight into the pros and cons of using OCaml for a major systems software project.

3. The team that used OCaml was one of five teams working on XenServer. This enables us to draw comparisons between the OCaml team and other teams within the XenServer group.

We start with a brief background into the XenServer engineering group (§1.1) and the product (§1.2). Next we describe the authors' perspectives of using OCaml for the XenServer project, reflecting on both our technical experiences and the different reactions within the company that we encountered regarding the use of

a non-mainstream language for product development (§2). In the remainder of the paper we present data that compares the OCaml team with other XenServer teams, in terms of hiring patterns (§3) and code contribution (§4). Finally, we examine other work in the community (§5) and conclude (§6).

1.1   The XenServer Engineering Group

The XenServer engineering group is organised into five separate engineering teams, each responsible for different software components that comprise the XenServer product. There is a Hypervisor/Kernel team, a Storage team, a Management Tools team (MTT), a Windows Driver team and a User Interface team. Four of these teams use "mainstream" languages (including C, Python and C#), but the MTT use OCaml as their primary development language.

There are about 40 engineers in total within XenServer engineering, including 10 full-time OCaml programmers in the MTT who are responsible for extending and maintaining a code-base that consists of approximately 130 KLoC of OCaml. The MTT team's components consume and provide interfaces and APIs to those of all other teams; thus there is constant interaction between the OCaml programmers and the rest of the development group.

1.2   The XenServer Product

Citrix XenServer is a managed virtualisation platform built on the open-source Xen hypervisor [1], offering a range of additional management features. Some of the features include:

Resource pools: The ability to create clusters of servers and shared-storage that are managed as a unit. Virtual Machines (VMs) can be moved between servers in the pool while continuing to run [3].

High Availability (HA): The ability to restart VMs on other servers automatically, if the server they were executing on fails. Cluster fencing, required to preserve data integrity in the storage layer [5], is provided in software by the XenServer management tools.

XenAPI: An XML-RPC management API that provides the ability to create resource pools and VMs, and configure all aspects of the system.

XenCenter Management Console: A Windows GUI that allows administrators to create and configure VMs and resource pools.

1.2.1   Architectural Overview

XenServer is based on a type-1 hypervisor [1], and is installed straight onto the bare metal and booted directly from a server's BIOS. The hypervisor is the first component to be loaded and

takes control of CPUs, memory and interrupt mappings. Next, it spawns a *control domain*—a small Linux VM that provides system management services and provides physical device drivers for networking and storage.

The main XenServer management process that resides in the control domain is known as XAPI, because it is the service that provides the XenAPI. The service's primary responsibility is to listen to XenAPI calls (made over the network) and execute these requests. In addition XAPI itself implements resource pools (dealing with the distributed systems challenges that this entails), maintains a durable, replicated persistent database of configuration data on behalf of the resource pool and is responsible for high-availability planning and failover[1]. The XAPI source code, consisting of approx. 130 KLoc of OCaml, is open source and can be freely downloaded under the LGPLv2 license[2].

One of the defining characteristics of XAPI is that it communicates with all major components of the system. On the one hand it accepts connections from clients (e.g. the XenCenter GUI), performing XenAPI requests on their behalf and providing access to a variety of data-streaming services (e.g. remote-access to VM consoles, importing and exporting VM disk images). On the other hand, XAPI interfaces with other software components within the server, including the Xen hypervisor and the networking and storage subsystems. This requires XAPI to use a variety of different interfaces, including (*i*) calling into statically-linked C APIs to communicate with the Xen hypervisor and the Linux kernel; (*ii*) forking new processes to invoke vendor-specific storage scripts or other shell commands; (*iii*) utilising a variety of different IPC mechanisms, for example to communicate with subprocesses involved in a live VM migration [3]; and (*iv*) performing protocol processing functions over both TCP and Unix domain sockets to receive and parse XenAPI requests.

Another property of XAPI is that it is highly concurrent. As well as managing a number of long-running background housekeeping threads, XAPI accepts and processes concurrent XenAPI requests across multiple connections from multiple clients and deals with communication between the multiple servers and shared storage devices that comprise a resource pool.

2. Authors' Perspectives

In this section we describe our perspectives of using OCaml within the context of the XenServer project. We discuss why OCaml was selected, describe the reactions within the company to using a non-mainstream language for product development and relate some of our technical experiences.

2.1 Selection of OCaml

The XenServer product did not start out within Citrix, but was first conceived within a startup called XenSource. Citrix acquired XenSource (and hence the XenServer team and product) in 2007. There were a number of factors within XenSource that drove the choice of OCaml and enabled the XAPI project to reach inception:

1. XenSource was staffed by a number of ex-researchers from the University of Cambridge Computer Laboratory. Many of these engineers had used OCaml before in a research environment and believed that, for large projects, the OCaml language offered significant productivity benefits over both traditional systems languages such as C, and dynamically typed languages, such as Python [10].

2. As a startup, XenSource had a culture of innovation and risk-taking. In this environment there were a number of influential people within the company who supported the use of OCaml, feeling that the risks of using a non-mainstream language were worth taking in return for the efficiencies that the engineers claimed it would bring.

3. XenSource had weak project governance within engineering. Thus, even though there were many people within the company who felt that using a non-mainstream language was not the right decision, the OCaml project started anyway and quickly built momentum as a grassroots effort.

These factors are all non-technical; they created the environment in which a product-development initiative based on a non-mainstream language could be seeded. But there were also technical reasons why OCaml was chosen over other languages for the XAPI project:

1. **Performance**: XenSource engineers had used OCaml on previous projects and were confident that it could deliver the required performance for the project [11].

2. **Integration**: OCaml's low-overhead foreign-function interface and existing Unix bindings facilitated the required interactions with the myriad of software components that made up the XenServer system.

3. **Robustness**: As a long-running service, XAPI must not crash. This requirement made OCaml's static type-safety and managed heap very appealing, offering the potential to reduce run-time failures due to type errors, memory leaks or heap corruption.

4. **Compactness**: there were plans for embedded versions of XenServer on flash storage as small as 16MB. The relatively simple OCaml run-time and compact native code output were key to this requirement.

There were other languages that met the above criteria, the most notable being Haskell. The primary reason for choosing OCaml over Haskell was non-technical. The engineers involved in the project had considerably more experience of using OCaml, and using it reduced training costs (this being a luxury in a fast-paced startup). Our previous experiences had also given us confidence that the OCaml tool-chain would meet the project requirements.

2.2 Reactions within the company

Choosing OCaml for a product development project was a contentious decision that created some heated debate within XenSource. While the engineers in the MTT firmly believed that the benefits of using OCaml outweighed the risks, others strongly believed that the risks of using a non-mainstream language for a major product development project were simply too great. Specific risks that were highlighted included:

1. We will not be able to hire OCaml programers quickly enough to grow the team.

2. A large code base in a non-mainstream language will make XenSource a less attractive acquisition target.

3. Other teams (staffed with programmers who don't know OCaml) will not able to work with the MTT because of "the language barrier".

4. The OCaml tool-chain may not be mature enough to support the development of a complex system.

The MTT had enough experience of using OCaml to argue convincingly that Risk 4 could be effectively mitigated. However, at the time the XAPI project was initiated, there was no data available regarding Risks 1—3, so debate (although heated) made little forward progress.

In hindsight, none of the risks above materialised. A year after work on XAPI started, Citrix paid $500M for XenSource, and the technical due-diligence process performed during the acquisition made it very clear that a large chunk of XenServer was implemented in OCaml. There were also no problems hiring OCaml programmers (§3), and other teams were able to work very effectively with the MTT (§4).

2.3 Technical experiences

We conducted a preliminary user study among the engineering group, with a set of open-ended questions designed to elicit individual opinions. Overall, the MTT report positive experiences of using OCaml on the XenServer project. Without exception, the engineers within the MTT believe that developing XAPI in OCaml has been a success, with the type system and automatic memory management being the most widely cited benefits of the language. Engineers also report that they *"enjoy programming in OCaml"*, particularly emphasising the fact that they believe OCaml allows them to express complex algorithms concisely. There is also a shared belief within the MTT that, overall, the choice of OCaml has enabled the team to be more productive than they would have been had they chosen a more mainstream language for the project (e.g. C++ or Python). Note that Java and .NET-based languages were not included due to the size of their runtime environments not being conducive to the 'compactness' requirement (§2). These positive experiences are backed up by internal test data and component defect levels that demonstrate that the quality and performance of the XAPI component is good.

However, despite the overall positive outcome, there have been some technical challenges that relate to the choice of OCaml. These challenges are not due to the OCaml language *per se*, but are due to lack of available library support, the complexity of the Foreign Function Interface (FFI) and the limitations of the OCaml toolchain. We consider each of these issues in more detail in the remainder of this section.

2.3.1 Lack of Library Support

We found that OCaml's library support for common data structures and algorithms generally sufficient for our needs. However, the lack of library support for common systems protocols was more problematic. In particular we ended up having to write a pipelined HTTP/1.1 server from scratch and handcrafting our own SSL solution using separate `stunnel`[3] processes to terminate and initiate SSL connections, and communicating with these over IPC.

There were some open source HTTP and SSL OCaml libraries available. However, at the time, the libraries that we evaluated were not fully featured or robust enough to meet the requirements of the XAPI project.

2.3.2 C Bindings

Writing C bindings was difficult and error-prone. Despite careful code-review and a policy of "keeping things simple" (avoiding references into the heap across the FFI, and avoiding use of callbacks whenever possible) some bugs still crept through, creating occasional XAPI segmentation faults that were hard to reproduce and track down.

2.3.3 Lack of Tool Support

Our heavy use of threads and *fork(2)* made it impossible for us to effectively use `ocamldebug` or `ocamlprof`. Instead we relied on `gdb` and `gprof` directly against the compiled binary. This was better than nothing, but the low-level nature of `gdb` made it hard to relate the debugging output back to the OCaml source.

Likewise, the lack of high-level profiling data made performance tuning harder than it should have been, and made it difficult to track down memory leaks[4].

2.4 Technical Lessons Learnt

Over the last four years of commercial OCaml development, we have learnt several technical lessons regarding its use. Some of these are outlined in this section.

2.4.1 Stability of Tools and Runtime

In the early days of XAPI development, we had no idea if the OCaml runtime (e.g. the garbage collector) would be robust enough to support long-running processes like XAPI that are required to execute continuously for months at a time. We joined the OCaml Consortium to offset this risk, providing us with a support channel in case bugs arose.

However, it transpired that the OCaml runtime was remarkably stable. Our automated test system puts XAPI through 2000 machine-hours of testing per night, and also runs regular stress and soak tests that last for weeks on end. Customers also run their XenServers for several months at a time without restarting XAPI. Despite all this testing, we have never had a single XenServer defect reported from internal testing or from the field that can be traced back to a bug in the OCaml runtime or compiler. (During development we did once find a minor compiler bug, triggered when compiling auto-generated OCaml code with many function arguments, but this was already fixed in the development branch by the time we reported it and so no interaction with the maintainers at INRIA was required.)

2.4.2 The Right Style for the Right Job

OCaml allows for many programming techniques to be used in the same codebase. XAPI takes full advantage of this fact, using different programming styles to solve different problems:

Imperative Many of the lower-level modules of XAPI (e.g. those that interface with the hypervisor and control domain kernel) consist of step-wise, imperative code and look like type-safe C. OCaml fully supports this style with language constructs such as `for`/`while` loops and references.

Functional Although a good chunk of XAPI is unashamedly imperative, some of the higher-level aspects of the system are functional in nature. For example the high-availability feature requires algorithms for distributed failure planning. These algorithms (e.g. bin packing) are implemented in a functional style.

One function of XAPI is to communicate with *Xenstore*. The Xenstore service, which runs in the control domain, provides a tuple-space that is used for co-ordination between VMs and the XenServer management tools [7]. Xenstore exposes an asynchronous event interface that is hard to use. XAPI abstracts much of this complexity behind a straight-forward combinator library that handles events via composable functions. For example, consider the following code fragment:

```
wait_for (any_of [
  'OK, value_to_appear  "/path1"
  'Failed, value_to_become "/path" v ])
```

The expression `value_to_appear "/path1"` represents the act of waiting for any value to become associated with key `"/path1"`. The expression `value_to_become "/path" v`

---

[3] Universal SSL wrapper: `http://www.stunnel.org`

[4] In a garbage collected language, like OCaml, memory leaks occur when global references to objects are not cleaned up explicitly (e.g. if something is added to a global hash-table and not subsequently removed).

represents the act of waiting for a specific value `v` to become associated with key `"/path"`. The expression `any_of` represents the act of waiting for *any* one of a set of labelled options; in this example the label `'OK` is used to represent a success case and the label `'Failed` represents a failure case. Finally the function `wait_for` uses the Xenstore event interface, returning either `'OK` or `'Failed` as appropriate.

Meta-programming XAPI has a distributed database that runs across all the hosts in a resource pool, including failover and replication algorithms. The OCaml code to interface with this database and remote calls is all auto-generated from a succinct specification and compiler. Similarly, all of the XenAPI bindings to other languages (C, C#, Java) are generated from a single data-model.

Object-oriented OCaml provides a comprehensive object system, but it is not used in XAPI except in small, local cases. Although we have nothing specific against using it, a compelling case for introducing them has never emerged. Modules, functors and polymorphic variants have been sufficient to date, and we anticipate that first-class packaged modules (in OCaml 3.12+) will further reduce the need for using objects.

2.4.3 Garbage Collect Everything

The automatic memory management that OCaml provides is a huge improvement over using C, but we still frequently get leaks due to mismatched allocation/deallocation of other limited OS resources, such as file descriptors and shared memory segments. These are usually only detected after automated stress testing detects the failure since the code involved works fine during development.

Nowadays, we make an effort to abstract as many of the OS resources as possible behind our own extensions to the standard library.

3. Hiring Patterns

Despite concerns raised at the start of the XAPI project, the MTT has had no difficulty in finding and hiring good OCaml programmers, and has been able to grow at a comparable rate to the other XenServer teams that used mainstream languages. From October 2006 to April 2010, 12 engineers have been hired into OCaml-programming positions (roughly a quarter of all XenServer engineers hired over the period).

There are two interesting observations about the MTT's hiring patterns. Firstly, we found that posting on functional programming mailing lists (including the OCaml List and Haskell Cafe) has consistently generated good inflows of high quality candidates interested in industrial functional programming positions. And, secondly, we have found that previous OCaml experience is not a prerequisite for hiring into OCaml-programming positions.

In fact, of the 12 engineers hired, only 2 had prior experience of OCaml; the other 10 learnt OCaml after they started work at XenSource or Citrix. Interestingly, having to learn OCaml did not make a big difference to the training time of the new engineers: the 10 engineers that did not know OCaml became productive at about the same speed as the 2 engineers that did have prior OCaml experience.

We believe that this is because, for a complex software product like XenServer, getting to know one's way around the various code-bases and getting to grips with the architectural principles of the wider system is a much more time consuming task than learning a new programming language. The 10 engineers that did not know OCaml were already highly proficient programmers who had a solid grounding in data-structures, algorithms and computer science more generally.

4. OCaml Code contribution

As described earlier (§1.1), the XenServer Engineering Group consists of five teams of full-time software engineers, supplemented by contractors. Each team is responsible for a different software component. The source code for each component is stored in a number of version-controlled repositories using Mercurial [14]. Each repository contains a complete historical record listing every code change, when it was made, who made it and why. In this section we will examine this historical record to identify and analyse which teams contributed to which components. We shall use this data to answer the question:

> *"Did the use of OCaml within the MTT prevent engineers from other teams making significant contributions to the XAPI project?"*

For our analysis we shall focus on four components:

1. Management Console: a windows user-interface maintained by the User Interface team;

2. Storage: a set of plugin modules to connect XenServer to back-end storage arrays where VM disks are stored maintained by the Storage team;

3. XAPI: the component which implements the XenAPI maintained by the MTT; and

4. Windows drivers: drivers required for high-performance VM I/O, maintained by the Windows Driver team.

The components were chosen for the following reasons:

1. they were all created solely for the XenServer product unlike, for example, the open-source Xen hypervisor that was created as part of a research project a few years before the XenServer product emerged;

2. they are all maintained by different teams; and

3. they all *primarily* use different programming languages (even the XAPI code contains traces of C).

The following table gives approximate sizes and primary language data for each component[5]:

Component	Size	Main Languages
XAPI	130kLOC	OCaml
Windows Drivers	80 kLOC	C, C++
Management Console	200kLOC	C#
Storage	40 kLOC	Python, C

The diagram in Figure 1 displays four bars, one for each component in the analysis. The height of each bar indicates the total number of individuals who contributed code to each component. The bars are subdivided into sections, each one coloured to indicate the team the contributor belonged to.

The diagram in Figure 2 displays four bars, one for each component as before. The bars now represent the relative *contribution size* from members of each team to each component. It is clear that, in all cases, the team responsible for maintaining a component makes the majority of contributions. However it is also clear that, in all cases, members of other teams made contributions.

The size and colouring of the bar corresponding to XAPI in Figure 1 clearly shows that the use of OCaml did *not* prevent engineers from other teams making contributions. Furthermore, the size and colouring of the bar corresponding to XAPI in Figure 2

---

[5] The XAPI number excludes auto-generated OCaml code, the Windows driver excludes header files as most are auto-generated, and the Management Console excludes auto-generated XenAPI and Windows Forms code.
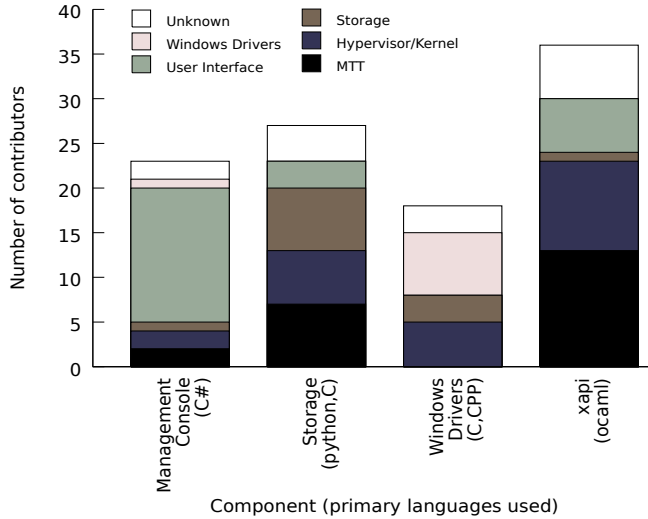
Figure 1. The total height of each bar shows the total number of unique contributors to each component. The color indicates the proportion of contributors from each team.
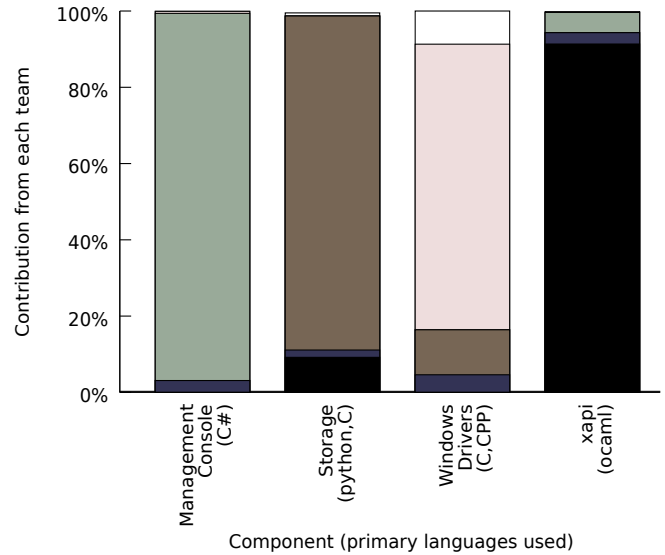

Figure 2. Each coloured section indicates the **size** of contributions to a component by a team, relative to the total contributions.

clearly shows that these contributions were as *significant* (in terms of size) as contributions made to other non-OCaml components.

5.   Related Work

There are several groups using OCaml in industrial settings. Jane Street Capital is a successful proprietary trading company which uses OCaml for a wide range of tasks. In their experience report [12], they share several of our technical concerns with OCaml: (*i*) generic pretty-printing facilities have to be addressed via macros; and (*ii*) the lack of a wide range of community libraries for common tasks. Since their report, some of these aspects have improved somewhat. OCamlForge provides a central place to locate community libraries, and systems such as *dyntype* [8] and *deriving* [16] make it easier to operate on generic values and types without modifying the core OCaml tool-chain. Like them, XAPI also does not use the OCaml object system much.

One concern we do *not* share is the lack of a multi-threaded garbage collector. Since XAPI is not a CPU-intensive service, and the control domain is limited to a single virtual CPU, the simplicity and stability benefits of the existing collector exceed the more complex concurrent alternative.

XenServer is not a hosted service, but a product that ships externally to many customers. MLdonkey [9] was one of the earliest (and for some time, the most popular) peer-to-peer client applications, written entirely in OCaml. We restricted our use of OCaml to the server-side component of XenServer, and wrote the native Windows client using C#.

We made some attempts to compile portions of the OCaml code (e.g. the command-line interface) for Windows, but the lack of robust libraries (particularly SSL) made it not worth the effort. Since our decision in 2006, desktop programming using functional languages has advanced considerably, as (*i*) Microsoft F# provides full access to Windows APIs [15]; and (*ii*) web browsers can host entire applications in Javascript, and be programmed in a functional style [2, 6]. We have not yet built a client using these technologies, however.

OCaml is traditionally popular as a compiler tool, and Frama-C is an example of an industrial-grade static analysis product [4].

XAPI also has compilers written in OCaml to generate bindings from an executable specification for more verbose languages like C#, C, Java and Javascript. This helped keep the various XenAPI clients synchronised with the server as it developed rapidly in the early days.

6.   Conclusions

The XAPI project is perceived as a success within XenServer engineering. The MTT works effectively with other teams (i.e. without any 'language barrier problems'), engineers have been hired into OCaml programming positions quickly and effectively and, technically, the XAPI component has shown itself to be stable and robust.

Although there were some drawbacks to using OCaml, namely a lack of library support for common protocols (e.g. SSL, HTTP) and a lack of tool support, engineers within the MTT believe that overall OCaml has brought significant productivity and efficiency benefits to the project. In particular, MTT engineers believe that OCaml has enabled them to be more productive than they would have been had they adopted one of the mainstream languages that would have met the requirements of the project (e.g. C++ or Python).

Since the XAPI code-base was open sourced in mid-2009 it has become possible for engineers beyond Citrix to work on the project. It remains to be seen whether the use of OCaml will act as a barrier to wider contribution, but based on our experiences reported in this paper, we are hopeful that it will not. We are already seeing some code submissions to the XAPI project from beyond Citrix and are working with development partners and the research community to encourage further contribution.

The source code can be obtained from `http://xenbits.xen.org/XCP/`.

7.   Acknowledgments

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] B. Canou, V. Balat, and E. Chailloux. O'Browser: Objective Caml on browsers. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 69–78, New York, NY, USA, 2008. ACM.

[3] 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.

[4] P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: OCaml for an industrial-strength static analysis framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 281–286, New York, NY, USA, 2009. ACM.

[5] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 175–188, Berkeley, CA, USA, 2003. USENIX Association.

[6] J. Donham. OCamlJS, July 2010. `http://jaked.github.com/ocamljs`.

[7] T. Gazagnaire and V. Hanquez. Oxenstored: an efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 203–214, New York, NY, USA, 2009. ACM.

[8] T. Gazagnaire and A. Madhavapeddy. Statically-typed value persistence for ML. In *Proceedings of the Workshop on Generative Technologies*, March 2010.

[9] F. Le Fessant and S. Patarin. MLdonkey, a Multi-Network Peer-to-Peer File-Sharing Program. Research Report RR-4797, INRIA, 2003.

[10] A. Madhavapeddy. Creating high-performance, statically type-safe network applications. Technical Report UCAM-CL-TR-775, University of Cambridge, Computer Laboratory, Apr. 2006.

[11] 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.

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

[13] T. Morgan. Citrix desktop virt soars in Q4, Jan. 2010. `http://bit.ly/ciB74a`.

[14] B. O'Sullivan. *Mercurial: the definitive guide*. O'Reilly Media, first edition, 2009.

[15] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*.

[16] J. Yallop. Practical generic programming in OCaml. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 83–94, New York, NY, USA, 2007. ACM.