Research Statement

Ali José Mashtizadeh

My research focuses on making computer systems reliable and easy to manage.

The past decade has seen a rapid acceleration in the development of new and transformative applications in many areas including transportation, medicine, finance, and communication. Most of these applications are made possible by the increasing diversity and scale of hardware and software systems. At one end of the spectrum, services spanning thousands of machines routinely compute over data sets of unimaginable size. At the other, consumer products including desktops, phones, cars and IoT devices have created a broad ecosystem of devices and sensors for developers to build on.

While this brings unprecedented opportunity, it also increases the probability of failures and the difficulty of diagnosing them. For example, a recent bug in the 787 avionics system [14] where all three flight control systems can reset simultaneously, led the FAA to issue guidance to periodically reboot these systems until a permanent fix can be found.

Increased scale and transience has also made management increasingly challenging. Devices can come and go for a variety of reasons including mobility, failure and recovery, and scaling capacity to meet demand.

Systems need to be able adapt to these conditions without human intervention. Operator error is a significant cause of outages and data loss, fundamentally limiting reliability. Further, many tasks—from automatically recovering from large scale failures, to seamlessly incorporating large numbers of mobile devices—are not possible with a human in the loop.

I have approached these challenges by building systems that decouple software state from physical hardware, and thus can adapt as hardware comes and goes.

While at VMware, I built several systems for *live migration*. Live migration decouples virtual machines (VMs) and disks from the underlying hosts and storage systems, allowing them to be moved to other devices at runtime without service interruption. This makes dynamic resource scheduling, zero downtime hardware maintenance and scaling, disaster recovery, and a variety of other automated management tasks possible.

The systems I built included live storage migration [7] that enables migrating storage without downtime, XvMotion [10] that enables migration of virtual machines and storage over longer distances, from crossing racks, to crossing the globe, and Centaur [12], a system that reworks the virtual disk abstraction to allow near instantaneous storage migration enabling real time IO scheduling.

At Stanford, I built Ori [8], a reliable distributed file system for devices at the network edge. Ori decouples files from any particular device, and automates many of the tasks of storage reliability and recovery through replication, taking advantage of fast LANs and low cost local storage in edge networks.

Most recently, I built Castor [17], a low-overhead multicore record/replay system. Castor decouples applications from hardware, allowing execution to be replicated across hosts. This enables transparent fault tolerance in network services. Castor also allows failures to be recorded when they occur in production, then later replayed for analysis, greatly simplifying the task of diagnosis.

Another approach I take to building systems is rethinking solutions at different layers of the stack, Castor is a good example of this.

Existing approaches to transparent fault tolerance work only at the hardware (Stratus) or virtualization (VMware FT) layer. Both have many limitations. Stratus requires specialized hardware, while VMware FT needs a fast dedicated local interconnect and adds non-trivial overhead. Both systems are very complex and required many years of effort to develop.

Castor provides a simpler and more capable approach by attacking this problem at multiple layers. It leverages compiler instrumentation, shared library and language runtime support, in conjunction with low level architecture features to implement record and replay as a basis for fault tolerance. It is very low overhead and requires no specialized hardware. Consequently, it can support fault tolerance in a much wider range of settings.

Much of my inspiration comes from solving my own problems. I have been collecting, fixing, and running computer systems for most of my life, as well as writing software for these systems. Ori was inspired by a desire for reliable and easily managed storage on my own devices. Castor was developed in response to my needs developing Ori. While building Ori, I was also frustrated with existing solutions for hardening my code against attacks. This led to a better system for mitigating C/C++ memory attacks called CCFI [13].

While the inspiration for these systems is personal, the solutions they offer have broad importance. Hardware architectures, compilers, operating systems, and the distributed systems we build on them are the foundation that future solutions in many other fields will rely on. The scale, diversity, and complexity of hardware is rapidly growing. Consequently, it is important that we continually re-assess how best to architect these foundational elements to ensure that future systems will be safe, dependable, understandable, and will allow us to make full use of their capacity.

Migration Systems

For several years I was the technical lead for migration at VMware. There are two types of migration. *Storage Migration* moves virtual disks between volumes on different storage devices including storage arrays, NAS, and local disks. *Virtual Machine Migration*, moves VM's (CPU/memory/device) between hosts. All of this takes place at runtime, ideally with no service interruption.

Migration enables live upgrades, scaling, and hardware maintenance. It also enables cluster and data center level resource scheduling for optimal CPU, memory, IO and power utilization. By the time I left VMware, migration was in use by ∼80% of enterprise customers.

I led the development of two new storage migration architectures that reduced downtimes for storage migration from minutes to well under a second, thus enabling *live storage migration*, i.e., migration with no visible service interruptions [7].

I also led the development of long distance live migration (XvMotion) that eliminated the dependence on local shared storage and networking, and enabled VMs to migrate between independent physical hosts whether in the same rack or in data centers on separate sides of the globe [10].

Finally, I led efforts to span the virtual disk abstraction across physical hosts and introduce storage load balancing in a system called Centaur [12]. This reduced total migration times from tens of minutes or hours to minutes or less, enabling responsive and efficient distributed IO scheduling.

Live Storage Migration The goal of live storage migration is to migrate a virtual disk between volumes without disrupting the services inside the virtual machine. Achieving this requires minimizing total migration time, minimizing downtime—the duration between pausing the VM on the source and resuming it on the destination, and minimizing the performance penalty induced on the workload during migration.

The original storage migration system could barely be considered "live," with downtimes in the range of minutes. Over the course of several years I iterated through two architectures for storage migration: an iterative copy approach tracking modified blocks (in ESX 4.0), and an IO mirroring approach (in ESX 5.0).

In both systems, I introduced several new optimizations to minimize downtime and total migration time. The final system brought the downtime down to predictably under a tenth of second, and minimized migration times to only slightly longer than the time it takes to copy virtual disks. I published a comparison of all three systems in USENIX ATC [7].

Long Distance Migration VMware's migration systems was originally built with the assumption of shared local storage, e.g., a SAN, and shared local subnet.

I led the design and implementation of a long-distance migration system that overcame these limitations called XvMotion [10], which first shipped with VMware ESX 6.0.

XvMotion enables migration between independent physical hosts, across racks, and even across data centers on different sides of the world. Decoupling VMs from the local network and storage array also enables new use cases, including test-to-production migration, whole data center maintenance, and whole site disaster recovery, e.g., tolerating the failure of a cooling system or other physical infrastructure that can cripple a data center.

XvMotion was a large effort that integrated the virtual machine memory and storage migration systems, and added a new application layer transport, in addition to changes in the ESX TCP/IP stack and Layer 2 virtualization. Our new transport supported both high bandwidth networking, e.g., multi-path over multiple 10 Gbps NICs, as well as increased tolerance for higher latencies on the WAN. To keep downtimes low over low bandwidth and high latency links, we introduced several new approaches to dynamically throttle workloads based on available bandwidth, ensuring graceful migration convergence.

Centaur: Virtualizing Virtual Disks A major use case for both live migration and storage migration is automated cluster level load balancing, in VMware's hypervisor this is performed by the VMware Distributed Resource Scheduler (DRS). The effectiveness of DRS is limited by how fast it can react to workload changes. Normal storage migrations can take anywhere from tens of minutes to hours, depending on virtual disk size and how many other heavy workloads a storage array is handling concurrently, thus preventing DRS from reacting to short term workload changes. Further, the IO overhead of storage migrations can severely impact other workloads on the same volume, impacting cost/benefit of DRS use.

A much more efficient approach is possible if we instead move from a virtual disk abstraction that is tied to a single volume to one that can span multiple volumes, potentially spanning multiple storage arrays or physical hosts. This allows DRS to migrate only the working set of the VM, usually only a few percent of total disk size. We enabled this by introducing a fine-grained storage load balancer at the granularity of virtual disk blocks.

While at VMware I mentored and collaborated with an intern through two summers to build such a system, which we called Centaur [12]. Centaur calculates miss rate curves and detailed statistics (e.g., IO size, read/write ratio) of VM IO workloads, then partitions virtual disks among multiple volumes, and precisely controls what percentage of the read or write working set to move between volumes. The result was a migration system with vastly lower migration overheads and the ability to dynamically adapt scheduling of migration IO based on cluster level IO scheduling policy. Rather than migrations taking an hour, Centaur can react to workload changes within minutes.

Ori File System

When I came to to Stanford for my PhD, I wanted to build a storage system that would support reliability for storage devices beyond the data center. Unlike my work at VMware, I was not confined to working within the virtual disk abstraction or proprietary storage arrays, and could reconsider storage abstractions.

To address this, I built Ori [8]. Ori replicates user data intelligently across the pool of available devices, including laptops, servers, phones, etc. to ensure reliability. Ori applies the idea of decoupling software state from the underlying hardware through storage replication. This ensures that data is not tied to any particular device, thus is not lost if a device is lost, damaged, or stolen.

Cloud storage systems, e.g., Dropbox, provide centralized user storage in the cloud, a simple approach to reliable and easily managed storage. However, this approach also has limitations.

The size of physical disks in personal computers is growing much faster than the Internet bandwidth available to users. The time to transfer a typical disk over a typical WAN connection grew from 14 hours in 1990 to 278 days in 2013. Local area networks are typically two orders of magnitude faster than the WAN. In terms of cost, Dropbox commanded a 25x premium over local storage in 2013. Finally, placing one's data into the hands of a third party introduces security and privacy concerns.

Ori provides an alternative approach to enabling reliability and ease of management by automating replication, versioning and sharing on a local file system that can operate across devices at the network edge. In Ori, replication replaces backup as the primary form of reliability. User's can take snapshots and access older versions of files. Sharing occurs through a novel mechanism called grafting that allows users to export and import any subdirectory with history across file systems. Discovery of nearby nodes, replication, and pruning old history are taken care of automatically by Ori.

Ori has thousands of downloads, and dozens of users actively contributing feedback, bug fixes, and improvements.

Practical Default On Record/Replay

While building Ori, I found myself wanting a simpler way to build services that would continue to work even as devices disappeared, either because of hardware failure, loss of power, or just loss of a network connection. This inspired my work on Castor.

Castor [17] is low overhead multi-core record/replay system that can be used as a foundation for building fault tolerance, or capturing and reproducing production bugs when they occur. For fault tolerance, Castor can run a replica of a server in parallel. Thus, even if the server's hardware fails, or otherwise disappears, the service will not be interrupted. In this way, Castor decouples computing from particular hardware for availability.

Castor's low overhead makes it practical to leave on by default in production, either for fault tolerance or bug catching purposes. Careful use of hardware in Castor results in a 10x or more increase in log throughput compared to similar record/replay systems, e.g., a server could potentially handle 10x more requests per second for the same total record overhead. Use of compiler and architecture level techniques allows Castor to record and replay unmodified applications, requiring only recompilation.

Castor provides a very low-overhead, simple, and flexible way for developers to transparently add fault tolerance to multi-core applications. In comparison, VMM based approaches to fault tolerance relying on replay have not been able to support multi-core [16], while other approaches based on fast checkpointing incur substantial complexity, and are limited to protecting machines in close proximity with a dedicated NIC to handle synchronization traffic.

Castor also provides a practical solution for recording and reproducing production bugs when they occur. Existing approaches have often incurred significant overheads when dealing with multi-core applications that make them impractical for coping with production workloads. Castor can also be used to enable reverse debuggers, dynamic program analysis tools (e.g., race detectors) and other systems where decoupling heavy weight offline analysis from fast online recording is useful.

Our current prototype supports applications written in C/C++ and Go. We are communicating with developers of several platforms for an initial release of Castor. Our eventual goal is to provide a common platform for record/replay that others can use in research and production, similar to the way that LLVM/Clang enables work on optimizations, static and dynamic analysis.

Cryptographic Control Flow Integrity

Buffer overflows and other types of memory errors are a critical source of vulnerabilities in systems written in C/C++. As a developer of these systems, I wanted a better approach to hardening my code against these attacks than what was offered by the state of the art. The outcome was CCFI [13], an improved form of CFI.

Control flow integrity (CFI) is a technique to prevent an attacker from taking control of an application's execution, even if they are able to corrupt its memory. It works by limiting control flow pointers, e.g., function pointers, return pointers, and method pointers, to pointing only to the programmer's intended sites. Past CFI systems classify pointers through static analysis into groups of valid targets for any given indirect jump. As a result of relying purely on static analysis, previous systems have been overly permissive, and attacks have been shown against all static CFI implementations.

Cryptographic CFI is a dynamic approach to CFI. Unlike previous systems, it can classify pointers based on dynamic and runtime characteristics in addition to static analysis. CCFI dramatically reduces the impact an attack can have on application execution, limiting the attacker to only actively used code paths. CCFI is implemented as a compiler pass in LLVM that inserts cryptographic code to generate and verify MACs of pointers.

Currently, CCFI incurs between 2%–20% overhead for various server workloads. However, a majority of this overhead is in protecting the return stack, rather than function pointers. Intel's planned hardware shadow stack mechanism [15] should bring this overhead down to just a few percent for future versions of CCFI.

Future Directions

Replicated Storage at Enterprise Scale

Many challenges remain in storage systems with significant heterogeneity in storage devices, networking, and geographic proximity. For example, companies with multiple primary offices and branch offices often use an ad-hoc combination of medium scale storage systems, e.g., NetApp or EMC storage arrays, with branch office caches. These setups become overly complex because of the variety of mechanisms employed, e.g., replication, caching, and backups.

A unified approach like Ori can potentially provide a better way of addressing this with commodity hardware. However, a variety of challenges need to be addressed to realize a solution.

Point-to-point networking does not scale well for large storage applications. I am interested in revisiting some of the ideas from the JetFile storage system [1] to create a multicast bulk transport that is practical for today's users. Such a solution should provide authenticated key exchange, encryption, and congestion control. It should also adjust how nodes join and leave the multicast group depending on available bandwidth and latency.

Replicating across geographically distributed sites also raises other questions around latency, scalability, and consistency. How do we efficiently enable users to share data across sites? How does the our data model impact scalability and latency? How should consistency change for geographically distributed storage?

Replicated storage systems also run into problems when individual users can control a host and thus override security by impersonating users or modifying the file system directly. Grafting in Ori shares subdirectories in an all-or-nothing manner to avoid this entirely. I want to develop a more granular approach that can address the needs of a replicated storage system using cryptographic primitives.

Rethinking Reliability: Current storage systems have fixed configurations for storage reliability, i.e., a RAID level, based on recommended best practices that are updated every few years by major vendors. This approach delivers inconsistent reliability, and makes it difficult to dynamically scale storage.

Instead of the current static approach, storage systems should dynamically change their storage configuration (replication, placement, parity, etc.) as a function of policy and available resources, automatically bounding the probability of data loss. Better factoring in drive age and device type can lead to more precise models for failure prediction. Actively recomputing failure probabilities can keep reliability models up to date as storage configurations dynamically change.

Correlated failures are another area I would like to explore. Many systems ignore correlated failures of disks [2], [3], correlated failures of sector ranges [5], and drive health reporting (e.g., SMART) [2]. For example, traditional RAID5/6 stripes across the same sector on all drives, making it susceptible to correlated failure of sector ranges. Factoring storage diversity (e.g., brand, model, production run, and firmware version), and exogenous factors like power and cooling, into replication choices, could also be interesting.

Record/Replay: Mechanism to Applications

The capabilities that Castor provides opens the door to exploring several new avenues.

Transparent Failure Tolerance: My hope is to build a system where robustness to a wide range of serious hardware failures could be enabled simply and correctly by doing little more than recompiling and relinking applications. I think this could be very valuable in a wide range of settings, from traditional IT where downtime is costly, to industrial and medical settings, where continuous operation is critical.

Existing state machine replication approaches require building or refactoring applications into two parts, the part that is replicated and the part that is not. This is labor intensive and error prone, as the interaction of these parts must still be correct. A much simpler solution is provided by fault tolerance systems that replicate entire applications, such as VMWare's FT, Stratus, Castor. These, however, can only tolerate the failure of a single host.

An alternative I want to explore is applying consensus to the replay log itself [9], [11]. This can potentially enable applications to tolerate network partitions, multi-node failures, etc. with the same ease and transparency Castor provides for fault tolerance today. Beyond this, I would like to look at applying practical byzantine fault tolerance techniques in the context of Castor.

Automatic Triage: Everyone has had their browser or other application crash, resulting in a bug report being sent to the developer. Unfortunately, diagnosing the cause of crashes today is often manual and time consuming (if possible at all). Consequently, at many companies, including VMware and Microsoft [6], many bugs reported by customers are closed without ever being diagnosed or fixed. Even the problem of establishing if two bugs are the same can be a daunting task.

Record/replay provides a promising approach to addressing this challenge. Replay can provide essential context to support automatically classifying and diagnosing many bugs, with little or no manual intervention [4]. Prior work has just scratched the surface of what is possible. I am interested in looking at how we can apply replay, dynamic analysis, and program slicing techniques to making automatic triage and bug diagnoses a regular part of the development process.

References

[1] B. Grnvall, I. Marsh, and S. Pink, "A Multicast-based Distributed File System for the Internet," in *Proceedings of the 7th workshop on acm sigops european workshop: Systems support for worldwide applications*, ser. EW 7, Connemara, Ireland: ACM, 1996, pp. 95–102. [Online]. Available: http://doi.acm.org/10.1145/504450.504469.

[2] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure Trends in a Large Disk Drive Population," in *Proceedings of the 5th usenix conference on file and storage technologies*, ser. FAST '07, San Jose, CA: USENIX Association, 2007, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267903.1267905.

[3] B. Schroeder and G. A. Gibson, "Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You?" *Trans. storage*, vol. 3, no. 3, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1288783.1288785.

[4] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou, "Triage: Diagnosing Production Run Failures at the User's Site," in *Proceedings of twenty-first acm sigops symposium on operating systems principles*, ser. SOSP '07, Stevenson, Washington, USA: ACM, 2007, pp. 131–144. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294275.

[5] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An Analysis of Data Corruption in the Storage Stack," *Trans. storage*, vol. 4, no. 3, 8:1–8:28, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1416944.1416947.

[6] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (Very) Large: Ten Years of Implementation and Experience," in *Proceedings of the acm sigops 22nd symposium on operating systems principles*, ser. SOSP '09, Big Sky, Montana, USA: ACM, 2009, pp. 103–116. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629586.

[7] A. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai, "The Design and Evolution of Live Storage Migration in VMware ESX," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'11, Portland, OR: USENIX Association, 2011, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002195.

[8] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazires, "Replication, History, and Grafting in the Ori File System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 151–166. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522721.

[9] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the Speed of Multi-core," in *Proceedings of the ninth european conference on computer systems*, ser. EuroSys '14, Amsterdam, The Netherlands: ACM, 2014, 11:1–11:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592800.

[10] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin, R. Koller, T. Garfinkel, and S. Setty, “XvMotion: Unified Virtual Machine Migration over Long Distance,” in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC’14, Philadelphia, PA: USENIX Association, 2014, pp. 97–108. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643645.

[11] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, “Paxos Made Transparent,” in *Proceedings of the 25th symposium on operating systems principles*, ser. SOSP ’15, Monterey, California: ACM, 2015, pp. 105–120. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815427.

[12] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, “Centaur: Host-Side SSD Caching for Storage Performance Control,” in *Proceedings of the 2015 IEEE International Conference on Autonomic Computing*, ser. ICAC ’15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 51–60. [Online]. Available: http://dx.doi.org/10.1109/ICAC.2015.44.

[13] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazires, “CCFI: Cryptographically Enforced Control Flow Integrity,” in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15, Denver, Colorado, USA: ACM, 2015, pp. 941–951. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813676.

[14] Dominic Gates, *FAA orders Boeing 787 safety fix: Reboot power once in a while*, http://www.seattletimes.com/business/boeing-aerospace/faa-orders-787-safety-fix-reboot-power-once-in-a-while/, 2016.

[15] Intel Corporation, *Control-flow Enforcement Technology Preview*, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2016.

[16] VMware, Inc., *VMware vSphere 6 Fault Tolerance: Architecture and Performance*, http://www.vmware.com/files/pdf/techpaper/VMware-vSphere6-FT-arch-perf.pdf, 2016.

[17] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, “Castor: Towards Practical Default-On Multi-Core Record/Replay,” in *Proceedings of the 22nd international conference on architectural support for programming languages and operating systems*, ser. ASPLOS XXII, Xi’an, China: ACM, 2017.