

Kaushik Veeraraghavan

Research Statement

My research investigates new approaches that make it easier to build, deploy and use software systems. My doctoral research addresses two problems: improving the reliability of multiprocessor execution, and devising storage systems for mobile computing. My work has spanned across multiple research areas including operating systems, binary instrumentation, file systems, mobile systems, security and more recently, distributed systems. I enjoy learning about new research areas within software systems, and have leveraged insights from each of the above areas in my research.

In my thesis, I propose a novel runtime system that reduces the overhead of dynamic analysis (like data race detection) for shared-memory multithreaded programs by an order of magnitude. I demonstrate the practicality of my runtime system by using it to build the fastest guaranteed deterministic replay system and the fastest data race detection and survival system for commodity multiprocessors. Earlier, I proposed several storage systems that incorporated first-class support for mobile computing so users could easily access and update their data from any device, in any location, without worrying about data fidelity, consistency or replication.

As an experimental computer scientist, I validate my designs by building robust implementations and evaluating their performance in deployment. I have published papers describing my research in many of the top systems conferences and have been recognized with best paper and presentation awards at several venues (ASPLOS 2011, FAST 2010, OSDI 2006, HotMobile 2008). In the rest of this statement, I summarize my previous research and describe my future interests.

Thesis Research

Despite rigorous testing, multithreaded programs continue to be plagued by concurrency bugs resulting in security violations, performance degradation, program crashes and other incorrect behavior. The systems community recognizes this problem and has proposed several tools and techniques to improve program reliability. For instance, deterministic replay systems record a program's execution so buggy runs can be analyzed offline. Similarly, data race detectors can be used to identify harmful data races during development prior to shipping software. Unfortunately, while these tools and techniques run efficiently on a uniprocessor, they are impractical on a multiprocessor as they often slow down program execution by an order of magnitude or more. My doctoral thesis developed methods to lower the overhead of techniques like deterministic replay, data race detection and data race survival so it is easier to build and deploy multithreaded applications.

Deterministic replay. I started off by observing that deterministic replay is simple and effective on a uniprocessor but slow on a multiprocessor. As the vast majority of a program's execution is deterministic, recording a uniprocessor execution turns out to be very efficient as only rare operating system events such as interrupts, system calls and signals need to be logged and reproduced. If the program is multithreaded, it is still the case that only one thread runs on the processor at any given instance so the order of shared memory updates can be recreated by logging and reproducing the schedule with which threads are context-switched on the processor. On a multiprocessor, however, shared-memory accesses add a high frequency source of non-determinism—logging and replaying these accesses can drastically reduce performance.

In my dissertation, I propose *uniparallel execution*, a new paradigm for executing multithreaded programs on a multiprocessor. Unlike traditional multiprocessor execution that scales performance

by running each thread of the program on its own core, uniparallel execution scales performance by running multiple time intervals (*epochs*) of the execution in parallel. This is challenging as I need to be able to predict future program state if I am to run future time intervals in parallel with prior intervals. The insight in uniparallel execution is that I can run a second instance of the program speculatively to generate this future state. By constraining each epoch to a single processor, I benefit from the simplicity and low recording overhead of logging the uniprocessor execution. Additionally, I can achieve the speed and scalability of running on a multiprocessor by running multiple epochs concurrently. The cost of uniparallel execution is the increased hardware utilization of redundant execution. I believe this is reasonable as most applications cannot fully scale to utilize all available processors (for instance, the application could be bottlenecked on memory bandwidth); uniparallel execution allows us to cheaply repurpose spare hardware resources to improve program reliability.

I used uniparallel execution to build DoublePlay [4], the fastest guaranteed deterministic replay system for commodity multiprocessors. Since DoublePlay logs the uniprocessor execution, it can guarantee that all data races encountered during recording will resolve in the exact same way during offline replay. Additionally, since uniparallel execution is supported by the operating system, applications can transparently benefit from deterministic replay without any code modification. I presented this work at ASPLOS 2011 where it received the best paper award.

Detecting and surviving data races. While guaranteed deterministic replay systems can help us analyze and debug concurrency bugs after they occur, we can further improve application reliability by detecting bugs early during development prior to shipping software, or by automatically surviving unknown bugs at runtime. With this in mind, I next focused on building a system to detect and survive data races.

A data race occurs when two threads concurrently access the same shared memory location without being ordered by a synchronization operation, and at least one of the accesses is a write. Ideally, a data race detector should possess three properties: high coverage (i.e., find harmful data races), accuracy (i.e., report no false positives) and low overhead. Unfortunately, current data race detectors have to consider fundamental tradeoffs, such as restricting themselves to managed code, or sampling which decreased coverage, to achieve low overhead. This is because current race detectors analyze the events executed by a program—for a multithreaded program running on a multiprocessor, this implies tracking high frequency shared memory accesses.

To address this problem, I proposed *outcome-based race detection*, an alternate approach to detecting data races by running multiple replicas of a program and analyzing their output and program state for a divergence. To ensure that divergences are not false positives, all replicas see the same program inputs and same happens-before ordering of synchronization events and system calls. Additionally, to increase the likelihood that a harmful data race does cause the replicas to diverge, I proposed *complementary schedules* that ensure that replicas execute unordered instructions in opposite orders. By definition, a harmful data race will cause one of the replicas to execute the instruction ordering that leads to an incorrect outcome identifying the data race, while the second replica runs correctly allowing the application to continue execution past the data race.

I implemented my ideas in Frost [2], a data race detection and data race survival system. My strategy in implementing complementary schedules was to restrict each replica to a single processor and explicitly control the thread schedule. While effective, uniprocessor execution is inefficient as it cannot scale to utilize additional cores. To scale performance, I leveraged uniparallel execution to run 3 replicas of an execution: the leading replica runs ahead and provides a log of non-deterministic events while two trailing replicas run with complementary schedules and verify the correctness of the

leading replica. My evaluation demonstrated that Frost is substantially faster than a traditional vector clock-based dynamic data race detector for unmanaged code while achieving equivalent coverage. In addition, Frost can survive unknown data race bugs that manifest at runtime by diagnosing replica outcomes and initiating recovery. I presented Frost at SOSP 2011.

Previous Research

Prior to reliable multiprocessor execution, I worked on the problem of simplifying how users access and manage their data from mobile devices. Specifically, I observed that despite the widespread adoption of mobile computing in the past few years, our data management systems have not kept pace with emerging technologies forcing users to manually deal with the complexities of data replication, consistency and availability. I demonstrated that we can greatly simplify and enhance the user experience by incorporating first-class support for mobility and context-aware computing in storage systems.

Context-aware data management. Laptops, tablets and cellphones are increasingly replacing desktop computers as the primary clients of a storage system. These clients vary greatly in their platform, screen size, battery lifetime, storage and network capabilities. The systems community has responded to this development by proposing several adaptation systems that transcode data to meet the screen size constraints of mobile devices, reduce data fidelity to meet network bandwidth and energy constraints, etc. The primary drawback of these adaptation systems is that they build both an adaptation scheme that transforms the data, and a storage system for managing the different representations. As a result, each adaptation system exists as a monolithic implementation that requires custom changes to the operating system, the application and even the use of intermediary proxies. This construction hurts interoperability and hinders the adoption of adaptation schemes.

I addressed this problem by proposing the quFile [3] abstraction for context-aware data management. A quFile is a new file system object that encapsulates different physical representations of the same logical data. When the quFile is accessed, a data-specific policy takes into account the access context, such as the application requesting the data, the device’s screen size, battery status, etc. and selects the best representation to return. By evolving the file system to recognize context and manage multiple representations of the same data, I allow the developer to focus on building adaptation schemes. I demonstrated the generality of quFiles by implementing six case studies including resource management, data redaction, copy-on-write versioning and application-aware adaptation. By reusing existing directory support to encapsulate representations, I was able to add support for quFiles to an existing file system in about 1500 LOC while imposing a performance overhead of less than 1%. I presented quFiles at FAST 2010 where it received the best paper award.

Fidelity-aware replication. Many replication systems have been developed to provide a consistent view of data across a user’s devices. Unfortunately, these replication systems assume that all interested devices consume data of the same fidelity. This assumption is false on constrained mobile devices—for instance, a user’s cell phone might store only a subset of address book fields available on a desktop. During an internship at Microsoft Research Silicon Valley, I designed and built Polyjuz [6], a new fidelity-aware replication system that performs fidelity reductions and update reintegrations to ensure consistency as various devices exchange data. At the end of my internship, the Microsoft Sync Framework product group took the highly unusual decision to invite me to Redmond to present my research to their technical team, and my system implementation was demo’d to attendees of MSR’s TechFest 2009. I presented PolyJuz at MobiSys 2009 where it was forwarded to IEEE TMC as one of the top 5 papers in the conference.

User-centric authorization for distributed storage. Current authorization systems limit data access to a small set of devices that have been explicitly authorized by the user. This is problematic as the user cannot use a friend’s device, or a public computer in an internet cafe, to access his data. I address this in Cobalt [5], a user-centric authorization system. Cobalt encrypts data before storing it in a distributed data store and retains the encryption key on a user’s mobile device. Data can be easily retrieved from any computer, but can only be decrypted and used when the user’s mobile device is within physical proximity. I presented Cobalt at FAST 2007.

Future Research

I intend to make it easier to build and deploy software systems in datacenters, and to make it simpler to manage data on public and private clouds.

Automated bug detection and patching. One goal of fault-tolerant systems like Frost is to mask the harmful effects of a bug so the system can continue running. Despite its usefulness, fault-tolerance imposes a performance penalty or hardware utilization cost that hinders its adoption in large scale deployments like datacenters. One reasonable compromise is to use sampling so the fault-tolerant capabilities are only enabled when the machine load is low, or on some small subset of all available machines. This raises an interesting question: if the fault-tolerant system survives a harmful bug in one of the sampled machines, can we automatically derive a patch so the remaining production machines can also survive this harmful bug?

My insight is that we can capture the execution state on the surviving machine and reproduce it on other machines to derive the same benefit. For instance, Frost runs redundant replicas with complementary schedules to survive data race bugs. One could devise a system that analyzes the instruction ordering in a valid replica and uses binary instrumentation to reproduce this ordering on other machines. Note that this work builds upon previous research on live-update systems with the primary difference that these previous systems require patches to be manually developed while I propose generating patches automatically. There are several unanswered questions in this work including, (1) how should the execution state be captured and reproduced, (2) what is the performance impact of applying patches on a live system, and (3) can we re-use existing live patching systems (e.g., is it possible to automatically translate a given instruction ordering to a patch and reuse existing infrastructure such as LOOM [7])?

Improving datacenter performance via speculative execution. In my current post-doctoral work at Facebook, I am investigating the performance of geographically distributed services with the end goal of devising alternate designs for future Facebook datacenters. Broadly, Facebook datacenters are split into a small number of geographical regions. Each datacenter has multiple tiers consisting of a user-facing web server tier, intermediate application service tiers and a database backend tier. To ensure consistency in the presence of concurrent updates, all writes are redirected to the sharded master databases in one region. This architecture presents a challenging tradeoff: synchronous writes impose a performance penalty but guarantee correctness while asynchronous writes perform well but could result in data loss on a failure.

My insight is that I can apply external synchrony [1] to improve datacenter performance as the guarantee of synchronous execution only needs to be provided to external clients issuing requests to the web tier. By extending the boundary of the system to include the entire datacenter, I can speculatively overlap synchronous writes with processing in the service tiers, so long as user-visible responses are buffered by the web tier until the underlying speculations have committed. More

generally, I believe that speculative execution within the datacenter can improve not just the DB write performance, but also reduce the cost of synchronous backup/replication for fault tolerance, allow for more dynamic load balancing and parallelize expensive dynamic analysis (like security checks, spam analysis, privacy filters, etc.). There are several challenges to realizing this design including (1) can speculative execution be incorporated in a lightweight manner with minimal system modification to simplify adoption, (2) how can we efficiently track writes across multiple tiers (3) how should the ordering of writes be maintained, (4) in case of a failure, how should the speculations be rolled back?

Data management for home users. I increasingly use different web services to store the primary copy of my data. For instance, I store my photos in Facebook, my videos in YouTube, my email in GMail, my documents in Google Docs and miscellaneous files in Amazon S3. However, I don't necessarily want to store my only copy of data in these services as I do not fully trust them or wish to ensure additional reliability (e.g., from service outages). Similarly, I could choose to cache data locally for use when disconnected, in the presence of weak/poor network connectivity or expensive data service charges. I might also want a backup replica (e.g., I could replicate a photo album to both Flickr and Facebook so my friends can view my data on the service of their preference).

Unlike previous systems, this scenario does not provide a centralized server that retains the primary copy of a user's data. Hence, ensuring consistency in the face of concurrent updates at different replicas is challenging. My insight is to split the role of a server: I intend to retain a centralized server on the control path (so updates and invalidations can be correctly propagated), but serve data independently from the most appropriate replica. When a client requests data, the server directs it to one of the replicas (e.g., a local device, a network-attached disk, a particular web server etc). Additionally, since the server might not be aware of a client's particular access constraints (e.g., its battery status or network connectivity preferences), the server can return a list of alternatives to the client so the client can optimize for connectivity/cost/energy etc. There are several challenges in realizing this design including (1) how should the server be designed, (2) how should updates be propagated and conflicts resolved, (3) how should clients communicate with the server and fetch/store data on a replica, (4) how should clients optimize for local constraints?

References

- [1] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006).
- [2] VEERARAGHAVAN, K., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal, October 2011).
- [3] VEERARAGHAVAN, K., FLINN, J., NIGHTINGALE, E. B., AND NOBLE, B. quFiles: The right file at the right time. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (San Jose, CA, February 2010).
- [4] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. DoublePlay: Parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Long Beach, CA, March 2011).

- [5] VEERARAGHAVAN, K., MYRICK, A., AND FLINN, J. Cobalt: Separating content distribution from authorization in distributed file systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (February 2007).
- [6] VEERARAGHAVAN, K., RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., AND WOBBER, T. Fidelity-aware replication for mobile devices. In *Proceedings of the 7th International Conference on Mobile Systems, Applications and Services* (Krakow, Poland, June 2009).
- [7] WE, J., CUI, H., AND YANG, J. Bypassing races in live applications with execution filters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation* (Vancouver, BC, October 2010).