**AISHWARYA GANESAN – Research Statement**

Computer systems underpin every modern application that we interact with today. When designing systems, one must often tradeoff strong guarantees for performance or vice-versa. For example, shared-memory multiprocessors either provide strong memory consistency or performance but not both. Similarly, local file systems often tradeoff crash consistency for performance. Likewise, databases must often choose between a strong isolation level and performance.

*My goal is to ensure that computer systems enable applications to realize both strong guarantees and high performance. My research so far has focused on resolving the tension between guarantees and performance in distributed storage systems. I rethink fundamental problems and design novel solutions to achieve this goal.*

Distributed storage systems power modern applications in today's data centers. These systems must offer strong guarantees such as *consistency* and *reliability* to applications. With consistency, applications expect to see a meaningful view of the data. Strong consistency avoids unintuitive behaviors, resulting in two benefits. First, it vastly simplifies application logic, reducing programmer burden. Second, by presenting a meaningful view of data, it avoids confusion for users interacting with the system. Unfortunately, these advantages come at a performance cost: strongly consistent storage systems incur high latencies and scale poorly. Weakly consistent systems, in contrast, deliver high performance but impose a significant burden on the programmer to reason about data inconsistencies. It also leads to confusing semantics for end users. To address this problem, I build new distributed storage systems [**FAST20**, **SOSP21**] that provide strong consistency guarantees yet also perform well (§1).

With reliability, applications expect data to be safe despite hardware and software failures. However, similar to strong consistency, reliability against failures also often comes at the cost of performance. For instance, a system that synchronously forces updates to the storage device offers excellent resiliency to system crashes and power failures but incurs significant performance overhead. Similarly, protecting against storage failures (e.g., disk corruptions) usually involves costlier mechanisms such as rebuilding the data onto a new node. In my work, I build new mechanisms [**OSDI18**, **FAST18**] that improve robustness to failures while incurring little performance overhead (§2).

Apart from addressing the correctness vs performance tradeoff, I have also built tools to reason about correctness in distributed systems (§3). These tools have revealed several bugs in distributed storage systems [**OSDI16**, **FAST17**] and applications deployed on Kubernetes [**HotOS21**]. My work also improves performance in local storage systems by leveraging emerging technologies [**SYSTOR19**] and recent advances in machine learning [**OSDI20**].

With the computing landscape rapidly changing, distributed systems in the future must adapt to new hardware (e.g., smartNICs, computational storage devices) and emerging deployment scenarios (e.g., the edge). In my future work, I intend to build systems for such new hardware and deployment scenarios. My vision is to ensure that these next-generation systems satisfy the demanding requirements of modern applications while offering high performance. My experience with building consistent and reliable systems equips me well towards realizing this goal. I describe my plan to address these problems at the end (§4).

## 1 Consistency vs. Performance

I first describe two new systems that I built in my postdoctoral work at VMware Research and my Ph.D. thesis. These systems bridge the gap between strong consistency and performance. A key insight in both the solutions is that *the system can remain inconsistent as long as external entities do not observe the system state.* This idea allows the system to defer expensive work until state is externalized, leading to high performance while providing strong semantics.

**Exploiting Nil-Externalizing Interfaces.** Datacenter storage systems replicate data onto many servers for high availability. To ensure that the replicas remain consistent, they use a replication protocol (e.g., Paxos or Raft). These protocols require expensive coordination among the replicas to process a request. At a high level, a dedicated replica (called the leader) decides the order in which requests will be applied and replicates the requests to others. Once enough replicas respond, the leader applies the requests and returns the results. Such coordination is deeply ingrained in current protocols, incurring high overhead.

I instead observe that synchronous coordination can be avoided by paying attention to the interfaces of the storage system being replicated. In particular, I identify *nil-externality*, a storage-interface property that enables one to defer coordination. A *nil-externalizing* (nilext) interface may modify state within a storage system but does not externalize its effects or the system state immediately to the outside world. As a result, a system can apply nilext operations lazily, improving performance. Put in the key-value API is an example of a nilext interface; it does not return a result or an error (e.g., by checking if the key already exists). In fact, all updates (including read-modify-writes) are nilext in write-optimized key-value stores (like RocksDB) that form the basis of many distributed storage systems. Even when a system has non-nilext interfaces, applications use nilext interfaces far more frequently.

I built *Skyros*, the first replication protocol that takes advantage of nilext interfaces [**SOSP21**]. The key insight in Skyros is to *defer coordination until state is externalized*. Because nilext updates do not externalize state, they are made durable immediately (without coordination) by writing to many replicas, but expensive ordering and execution are deferred. The effects of nilext operations, however, can be externalized by later non-nilext operations (e.g., a read to a piece of state modified by a nilext update). Thus, nilext operations must still be applied in the same order across replicas for consistency. *Skyros* establishes this required ordering in the background and enforces it before the modified state is externalized. While nilext interfaces lead to high performance, it is, of course, impractical to make all interfaces nilext: applications do need state-externalizing updates (e.g., increment and return the latest value). Such non-nilext updates are immediately ordered and executed for correctness.

*Skyros* performs well in practice: real-world traces show nilext updates contribute to a significant fraction of writes, and reads do not often access recent updates. As a result, *Skyros* offers significantly lower latencies than traditional replication protocols such as Paxos while still providing linearizability. Overall, this work introduced a fundamentally new way to realize higher performance in strongly consistent replicated storage systems.

**Consistency-Aware Durability.** Some systems use weaker replication protocols that do not incur coordination overhead. Such a weak protocol achieves high performance by employing a weak, eventual durability model underneath. In particular, each update is immediately acknowledged after storing it on just one replica; the update is only lazily replicated to other replicas. However, eventual durability leads to weak semantics, exposing stale and out-of-order data. Thus, I ask: can a durability layer enable strong consistency yet also deliver high performance?

I designed *consistency-aware durability* (CAD), a new durability model that achieves this goal [**FAST20**]. CAD realizes that what *reads* observe is important. Thus, CAD shifts the point of durability to reads from writes. Because writes are delayed, CAD achieves the high performance of eventual durability. However, by ensuring that data is made durable before serving a read, CAD enables one to realize stronger consistency. In particular, I showed that *cross-client monotonic reads*, a new and strong consistency property, can be realized with high performance upon CAD. Immediate durability can enable this semantic, but not with high performance; this semantic simply cannot be realized upon eventual durability. By allowing reads at many replicas, our implementation of cross-client monotonic reads reduces latencies by order of magnitude compared to a strongly consistent system. Overall, CAD delivers high performance while enabling stronger semantics. In summary, my research demonstrates that by enforcing consistency requirements at a later point, stronger consistency can be realized without impacting performance.

## 2 Reliability vs. Performance

Protecting systems against failures, i.e., offering strong reliability, is also an important goal. However, strong reliability often comes at the cost of performance. In my work, I built systems that improve resiliency to failures while not impacting performance. I focus on two types of failures: crashes and storage failures.

**Resiliency to Crash Failures.** An important consideration in distributed systems is how each replica stores data. In the *disk durable* approach, critical state is replicated to persistent storage (i.e., disks). The disk-durable approach offers excellent durability and availability by committing to disks of many nodes. Unfortunately, disk durability is prohibitively expensive. As a result, most systems adopt *memory durability*, where the state is replicated only to the (volatile) memory. With memory durability, performance is high, but at a cost: durability. When many nodes crash, it can lead to data loss or unavailability. I built SAUCR [**OSDI18**], a *hybrid replication protocol* that provides the performance of memory durability while offering strong guarantees similar to disk durability. The key idea in SAUCR is that the mode of replication should depend upon the situation the distributed system is in at a given time. In the common case, SAUCR runs in memory-durable mode, achieving excellent performance; when nodes crash or become partitioned, SAUCR transitions to disk-durable operation, thus ensuring safety at a lower performance level.

**Resiliency to Storage Failures.** Each replica in a distributed system uses a storage device to store its data. These devices exhibit partial failures where a few blocks of data may occasionally be corrupted or remain inaccessible. Studies have shown the prevalence of such storage faults. However, most distributed systems handle such faults incorrectly, leading to data loss and unavailability. Thus, to improve the resiliency to storage faults, I designed *corruption-tolerant replication* or CTRL [**FAST18**]. The key idea in CTRL is to *co-design the local storage layer and the distributed protocols* to carefully use the inherent redundancy to recover from storage faults. CTRL significantly improves resiliency, offering safety and high availability while matching the high performance of unreliable counterparts.

## 3 Other Work

Apart from my primary research, I have built testing frameworks that identify vulnerabilities in distributed storage systems. I have also worked on solutions to improve the performance of local storage systems.

**Tools for Distributed Reliability.** Each replica in a distributed system depends upon a local storage stack to store data. However, little is known about how distributed storage systems react to problems that arise at the storage stack. I thus built two testing frameworks that examine these effects: PACE [**OSDI16**] and CORDS [**FAST17**]. PACE tests how local file-system crash behaviors impact distributed systems. CORDS examines how storage corruptions and errors affect reliability. Together, PACE and CORDS have discovered several vulnerabilities in popular systems and brought attention to a new class of failures that have been previously ignored. More importantly, these studies have shed light on fundamental reasons as to why today's systems are not resilient to storage problems.

**Leveraging Emerging Technologies.** Emerging technologies like persistent memory offer more capacity than DRAM, improving tail latencies of reads. They also provide persistence at lower costs compared to disks, improving write latencies. However, existing proposals to use such technologies can only improve either read or write performance but not both. We thus built PolyEMT [**SYSTOR19**], a system that dynamically configures available NVM capacity in volatile and non-volatile forms to improve performance for reads and writes simultaneously.

With faster storage devices, the cost of accessing data is getting cheaper. With increasing memory capacities, more data can fit in memory, reducing access costs further. Thus, searching data via an index contributes to a large fraction of the total query time. Learned indexes offer a way to reduce this cost. This approach uses a learned function to predict a data item's location, reducing costs. However, learned indexes in their original form do not allow index modifications. We built Bourbon [**OSDI20**], a system that applies learned index for LSM trees while allowing writes.

**Impact and Industry Collaboration.** My work has had significant practical impact. Our work on distributed storage reliability has exposed many severe bugs in popular distributed systems. A commercial product uses ideas from my research on corruption-tolerant replication to protect financial data against storage corruptions [link]. Follow-on work protects ZippyDB and HDFS at Facebook using an approach similar to CTRL [link].

At VMware Research, I am currently involved in three collaborative projects towards building resilient and performant systems. In one effort, we are exploring ways to integrate persistent memory into current key-value stores. A key question here is: how to perform this integration using a grey-box approach with no or minimal change to the applications? I am also working with other researchers to build a distributed write-optimized data structure. How do we build applications atop such a structure? How to ensure consistency and not compromise performance? Lastly, we are building tools to test the correctness of operators that manage applications on managed environments such as Kubernetes [**HotOS21**]. This tool has already found many severe vulnerabilities in operators, leading to data loss and unavailability. These projects often involve interacting with product groups to learn about their real problems and considering these learnings when developing the research systems. I expect these collaborations to continue and also enable my work to have a practical impact.

## 4 Future Research

I first outline my immediate-future plans where I will continue to exploit the insights gained in my prior work. In particular, I will discuss how I intend to improve the performance of distributed systems built using existing, popular architectures. I then sketch how I intend to realize my goal of building systems for emerging hardware and scenarios.

### 4.1 Existing Distributed-systems Architectures

**Interactive Client-server Applications.** Interactive applications (like virtual reality and gaming) built using the traditional client-server architecture often forgo fault tolerance for responsiveness. However, fault tolerance is desirable in many use cases; for example, a server failure leading to a complete loss of game state can be devastating. Applying the idea of deferred ordering and execution can alleviate this problem. In these systems, clients generate interactive inputs; the server then executes them, resulting in changes to the system state (e.g., moving of objects). The server then pushes the changes to clients, which render the view. At a high level, interactive inputs do not externalize state immediately; inputs are applied in some order by the server and later externalized. Interactive update interfaces, in essence, are nilext. Thus, ideas of nilext-aware replication can be used to make interactive servers fault tolerant without increasing latency. However, important questions must be addressed. For example, how to enable concurrent updates while preserving consistency? Can any replica push state changes for better scalability? More broadly, what other applications can benefit from deferred execution?

**Microservice Applications.** Modern applications are built using recent paradigms such as microservices. In the microservice architecture, a client request is processed in a chain of individual service components. This provides many benefits, such as modularity, scalability, and fault isolation. However, applications incur higher latencies because each component is internally replicated; thus, processing a request may require expensive coordination among the replicas. An extension to the deferred execution idea can help realize the benefits of microservices without compromising on latency. At a high level, if a request execution need not be finalized at a component, then the request can be simply

logged and later executed when external entities observe the component state in some way (e.g., when the state is explicitly read). Such deferral can lead to lower latencies and thus improve responsiveness. However, such a solution must address several challenges. For example, how do we efficiently track dependencies across components? How do we identify when coordination must be enforced?

**RDMA.** Remote direct memory access (RDMA) offers a way to communicate within data centers. RDMA does not interrupt the CPU at the receiver, enabling clients to access the server's memory directly. This leads to low latency and more scalability. However, such one-sided operations are not suitable (or inefficient) to implement general-purpose storage systems. For example, to read items indexed by a Btree, multiple one-sided operations are required. Prior work has designed solutions to perform reads using a single one-sided operation (e.g., by caching the internal nodes at the client). However, updates still involve the server CPU, incurring overhead for write-intensive applications. I realize that exploiting nil-externality enables one to process updates purely using one-sided operations. Because nilext interfaces do not return a response, they can be completed using one one-sided RDMA write. Such a solution can be valuable for microsecond-scale applications. However, realizing these benefits are not without challenges. For example, how can requests be ordered when the server CPU is not involved? It is worthwhile to rethink RDMA primitives and their capabilities and also leverage programmable RDMA NICs to solve such problems.

I look forward to establishing collaborations with researchers from other areas on these projects (e.g., with mobile-computing researchers on gaming and networking researchers on RDMA.)

### 4.2 Rethinking Distributed Systems for Emerging Hardware and Architectures

**New Hardware.** With new networking and storage technologies such as smartNICs and smartSSDs, we must fundamentally rethink how we build systems. These devices offer significant compute capabilities within the IO device, introducing possibilities to offload work to realize higher performance. I will take a key-value store as a concrete example. Key-value stores perform several background data-reorganizing tasks such as compaction; such jobs are not latency critical but help improve overall performance. With smartSSDs, background jobs can be pushed close to where data resides. Such offload enables the system to save host CPU work, leading to benefits. Similarly, smartNICs can help offload tasks in a distributed key-value store. For example, the memory in the smartNIC can be used to implement memtable-like structures common in many key-value stores. Such a design enables lower latencies because the NIC can process reads and writes independently without any CPU involvement. However, it is unclear how to extract the full benefits of these devices. For example, which tasks yield the highest benefits when offloaded? How do we carefully split the work and data to ensure application consistency? Can we build common abstractions (e.g., shared logs) within the devices so that multiple applications (such as databases and message queues) can benefit?

**New Scenarios.** In addition to new hardware, new deployment scenarios also necessitate rethinking systems design. A deployment setting that has been gaining traction is the edge. Traditionally, systems are deployed in the cloud, and clients access these services over high-latency links. With edge, compute and storage resources are available closer to the clients. Thus, the edge can improve the response times, which is important for interactive applications such as virtual reality, autonomous cars, and online gaming. The edge can also help reduce the bandwidth (by compressing or filtering the data). My goal is to build a framework that enables applications to leverage the infrastructure at the edge. However, building such a framework involves solving numerous challenges.

(1) How should *compute resources* at the edge be exposed? It is desirable to expose a high-level abstraction that reduces programmer burden. The serverless paradigm is an attractive option. However, it has limitations: high startup latencies and lack of support for stateful services. Designing programmer-friendly abstractions that enable one to realize the benefits of the edge is an interesting challenge. (2) How should *storage* be exposed? Will a key-value interface be sufficient? Or, must one provide transactions and complex data models to support a variety of applications? (3) *Fault tolerance* should seamlessly allow clients to connect to the service despite failures. How to make an edge service fault tolerant? How to keep data at the edge consistent despite failures? (4) *Migration* should allow clients to connect to different edge servers. How to migrate state and support strong consistency requirements? Can one leverage location and other sensors to predict movement and pro-actively migrate state? (5) How to *dynamically schedule* computations across the edge, cloud, and end device based on resource availability and SLA requirements? If resources are constrained at the edge, clients still should be able to access the service. (6) Finally, *security and privacy* is paramount for applications that need to protect their data from the provider and other tenants. Some applications may want to run analytics across all participating clients without leaking private data (e.g., federated learning). How can the edge infrastructure meet such security and privacy needs?

Overall, new hardware and deployment advances present new challenges for computer systems but also offer opportunities to innovate. I am excited to collaborate with researchers from other areas spanning networking, programming languages, security, and mobile computing to build these next-generation systems.

## References


[**FAST18**] Ramnatthan Alagappan, **Aishwarya Ganesan**, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-Aware Recovery for Consensus-Based Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018. (*best-paper award, invited for fast-tracked publication in ACM Transactions on Storage*)

[**OSDI18**] Ramnatthan Alagappan, **Aishwarya Ganesan**, Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[**OSDI16**] Ramnatthan Alagappan, **Aishwarya Ganesan**, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[**OSDI20**] Yifan Dai, Yien Xu, **Aishwarya Ganesan**, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From Wiskey to Bourbon: A Learned-index for Log-structured Merge Trees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Banff, Alberta, November 2020.

[**FAST17**] **Aishwarya Ganesan**, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017. (*best-paper award nominee, invited for fast-tracked publication in ACM Transactions on Storage*)

[**FAST20**] **Aishwarya Ganesan**, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Strong and Efficient Consistency with Consistency-aware Durability. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, February 2020. (*best-paper award, invited for fast-tracked publication in ACM Transactions on Storage*)

[**SOSP21**] **Aishwarya Ganesan**, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Nil-Externality for Fast Replicated Storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, Virtual, October 2021. (*invited for fast-tracked publication in ACM Transactions on Storage*)

[**SYSTOR19**] Iyswarya Narayanan, **Aishwarya Ganesan**, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. Getting More Performance with Polymorphism from Emerging Memory Technologies. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*, Haifa, Israel, April 2019.

[**HotOS21**] Xudong Sun, Lalith Suresh, **Aishwarya Ganesan**, Ramnatthan Alagappan, Michael Gasch, Lilia Tang, and Tianyin Xu. Reasoning about Modern Datacenter Infrastructures Using Partial Histories. In *Proceedings of the 18th USENIX Conference on Hot Topics in Operating Systems (HotOS'21)*, Virtual, May 2021.