Ravi Netravali — Research Statement

Web applications are integral to today's society, supporting a variety of services ranging from banking and e-commerce to navigation and social media. To keep pace with the evergrowing list of desired features and performance demands, these applications have evolved into complex dynamic systems. On the client-side, applications support rich user interactivity, numerous content formats (e.g., JavaScript, videos), and asynchronous IO patterns. On the server-side, applications are commonly distributed across many machines, and employ multi-tier architectures to implement application logic, caching, and persistent storage. As a result, tasks that were once straightforward for web applications, such as performance optimization and debugging, have become far more difficult. Such tasks are critical—delays of just milliseconds, or seemingly innocuous bugs, significantly affect user satisfaction, costing content providers millions of dollars in revenue.

My research addresses this growing complexity by identifying and analyzing the distributed data flows in web applications. Unlike prior taint-tracking and provenance systems which identify coarse-grained influences on program state, my work tracks fine-grained data *history* by scalably capturing precise execution traces and information flows, without requiring developer annotations. For instance, instead of logging that variable x has somehow been derived from y, my systems capture the exact list of code lines that accessed x. Tracking data flows at the granularity of individual pieces of program state, like JavaScript variables and DOM nodes on the client-side, and key/value pairs in storage systems on the server-side, provides invaluable insights into the low-level behavior of complex web services. Understanding that dynamic behavior enables more powerful performance optimizations and richer debugging tools.

The foundation of much of my research is Scout [2], a framework my colleagues and I built that rewrites web pages to automatically track fine-grained data flows. We have developed multiple systems that leverage Scout to improve various aspects of web services:

- Polaris [2] dynamically reorders client requests in a page load to minimize network round trips without violating data flow dependencies, reducing page load times by 34% (1.3 seconds).
- Prophecy [4] generates a snapshot of a mobile page's post-load state, which clients can directly process to elide intermediate computations that are normally discarded. Prophecy results in reductions of 21% in bandwidth consumption, 36% in energy usage, and 53% (2.8 seconds) in page load times. By using data flow graphs to guide the construction of snapshots, Prophecy provides numerous advantages over prior snapshotting techniques, including support for object caching and incremental interactivity.
- Vesper [5] is the first system that measures how quickly a page becomes interactive. Vesper determines a page's interactive state, which is not explicitly annotated, by finding and executing event handlers, and analyzing the corresponding data flows. User studies show that users prefer pages that optimize for interactivity. However, existing metrics inaccurately measure the time until a page is interactive by 39%. Optimizing for these metrics forgoes 32% of the benefits of targeting interactivity directly.
- Cascade [3] is the first replay debugger that makes fine-grained data flows explicit and queryable, enabling provenance tracking across multiple clients and servers. Cascade also supports speculative bug fix analysis, where a developer can replay a program to a certain point, change code in the program, and then resume replay to evaluate if the hypothesized bug fix would have helped the original execution.

My research interests are in building networked systems that have a positive impact on users. My approach is to identify and understand the fundamental problems causing frustration for clients and developers, and then create practical systems to overcome those problems. Examples of this approach include Mahimahi [6], which is widely used by industry and academic research groups, and Polaris, which is currently being deployed by several commercial content providers.

1 Tracking Fine-grained Data Flows in Web Applications

Scout: To capture the precise data flows in a web page load, we built Scout [2], a tool that automatically instruments a page's JavaScript and HTML to log fine-grained reads and writes to the page's state. Browsers do not explicitly expose all of the information flows in the JavaScript heap and the DOM (the browser's programmatic representation of the page's HTML tree). Thus, Scout must carefully model how browsers access that state in order to capture all of the data flows in a page load. Experiments show that Scout's tracking imposes only a 3%–5% overhead on page load times, making it usable in practice.

Polaris: A web page *dependency graph* captures the "load-before" relationships between a page's objects (e.g., HTML and JavaScript files), specifying the order in which those objects can be fetched and evaluated.

Browsers traditionally define these graphs solely in terms of initiation contexts. Unfortunately, that definition doesn't capture the true data flow dependencies between a page's objects. As a result, browsers must use *conservative* algorithms to fetch and evaluate objects, to ensure that hidden load-before relationships are not violated. For example, upon parsing the first of two adjacent HTML `<script>` tags, a browser must halt parsing of the page, and synchronously fetch and execute the referenced JavaScript object. The reason is that execution of the first object *may* create state that is read by the second object. Synchronously loading JavaScript files guarantees correctness, but is often too cautious; if two JavaScript objects do not modify mutually observable state, the browser should be free to load them in any order that reduces page load times.

Scout can identify the true data dependencies between a page's objects which govern the load process. Polaris [2] is a page load accelerator that allows unmodified browsers to exploit this information and avoid conservative assumptions about load order. When a client requests a page, the server responds with a Polaris scheduling stub rather than the page's HTML. The stub includes the page's original HTML and Scout dependency graph, as well as a small JavaScript library. This library runs a scheduling algorithm that uses the Scout graph and knowledge about already-fetched objects to determine which objects to load next using the browser's limited resources. By prioritizing paths in the Scout graph with the most unresolved nodes, Polaris minimizes the overall time needed to resolve the graph. Our experiments show that Polaris reduces median and 95th percentile page load times by 34% and 59%, respectively. Trial deployments of Polaris are underway at several companies, including American Express and Bounce Exchange.

Prophecy: Mobile browsing now generates more HTTP traffic than desktop browsing. On smartphones, 63% of user focus time, and 54% of overall CPU time, involves a web browser. Thus, mobile page loads are important to optimize along multiple axes: bandwidth usage, energy consumption, and page load time.

Prophecy [4] is a new acceleration system for the mobile web that improves all three aspects. With Prophecy, a web server analyzes Scout logs to determine the final value for each JavaScript variable and DOM object that is live at the end of a page load. When a mobile browser requests a page, the server returns a write log (with one entry per live value) instead of the traditional HTML, CSS, and JavaScript. By applying this write log, the mobile browser can elide slow, energy-intensive computations involving JavaScript execution and layout/rendering. Prophecy's write logs are smaller than a page's original content, and can be fetched in fewer RTTs. Prophecy's knowledge of fine-grained data flows also enables optimizations that are impossible for prior web accelerators. For example, Prophecy can generate write logs that interleave DOM construction and JavaScript heap construction, allowing interactive page elements to become functional immediately after they become visible to the mobile user. Our experiments reveal that Prophecy reduces median page load time by 53%, energy expenditure by 36%, and bandwidth cost by 21%.

Vesper: Modern web pages have sophisticated GUIs which support rich interactivity like user searches and animations. Unfortunately, existing web performance metrics do not precisely capture interactivity. The traditional "page load time" definition records when all of a page's objects have been loaded; this is overly conservative since only a subset of that state may be needed to allow a user to properly interact with above-the-fold content in the initial browser viewport. Newer metrics like Speed Index measure the time to render the initial viewport, but fail to capture whether the page has loaded critical JavaScript state for interaction.

To address these limitations, we created a new metric called *Ready Index* [5], or RI, which directly captures page interactivity. RI defines a page to be fully loaded when (1) the initial viewport has completely rendered, and (2) the JavaScript and DOM state that supports interactive functionality for each element in the initial viewport has been loaded. Existing systems are not equipped to measure RI since developers do not explicitly annotate the state that supports interactivity. To solve this problem, we built Vesper, a system that identifies a page's interactive state by finding and firing all of the page's event handlers, and analyzing the generated fine-grained data flows. Vesper measures RI by tracking the loading progress of interactive state and analyzing browser paint events to evaluate rendering progress. Our user studies show that users prefer pages that prioritize the loading of interactive content, motivating the need to measure and optimize for RI. However, our experiments reveal that prior metrics inaccurately measure the time until a page is fully interactive by 39%. Additionally, we used Vesper to extend Polaris to optimize for RI, and found that optimizing for prior metrics forgoes 32% of the RI benefits achieved by targeting interactivity directly.

Cascade: Debugging the asynchronous wide-area data flows in large-scale web services is difficult. Traditional debugging primitives like breakpoints can only crudely capture the temporal flows necessary to reconstruct a buggy value's provenance across multiple asynchronous codepaths. To fill this void, we de-

veloped Cascade [3], the first distributed replay debugger that makes fine-grained data flows explicit and queryable. Cascade provides three novel features to enable a fundamentally more powerful debugging experience. First, Cascade tracks *precise value provenance*, or the exact set of reads and writes that produce each program value. Developers can quickly query data flow logs to answer questions like "How did a user's mouse click affect variable z?" Second, Cascade enables *speculative bug fix analysis*. Developers can replay a program to a certain point, change code in the program, and then resume replay. Post-edit, Cascade replays nondeterministic events from a log using carefully-defined semantics about the context of the edited execution. Cascade also identifies divergences in data flow graphs to help determine the impact of hypothesized bug fixes. Third, Cascade supports *wide-area debugging* for applications whose server-side components use event-driven architectures like Node or Redis. The event loop interface provides a narrow, semantically-well-defined abstraction layer at which to log and replay. By tracking data flows across clients and servers, and by coordinating them using vector clocks, Cascade can reason about wide-area causality and support speculative replay of *distributed* applications. We have used Cascade to fix real bugs in complex applications.

2 Other Research

Pensieve [1] is a system that generates video streaming adaptive bitrate (ABR) algorithms using reinforcement learning. Unlike prior approaches that rely on preprogrammed (inaccurate) models of the video streaming environment, Pensieve learns an ABR control policy *purely through experience*. Pensieve uses observations of video QoE from past decisions to train a neural network model that maps "raw" observations (e.g., throughput and buffer data) to the bitrate decision for the next chunk. The neural network provides an expressive and scalable way to incorporate a rich variety of observations in the control policy, resulting in generalizable ABR algorithms. Our experiments show that Pensieve improves average QoE by 12%–25%.

Mahimahi [6] is a suite of tools that enables developers to record traffic from HTTP-based applications, and later replay recorded traffic under emulated network conditions. Developers can easily evaluate how new schemes across the stack (e.g., transport-layer protocols) affect application-layer metrics. Mahimahi's improvements stem from its UNIX shell-based design, which supports isolation and flexible experimentation. Mahimahi is open-source and is available as a Debian package. Further, Mahimahi is used by industry (e.g., Google, AT&T) and academic research groups (e.g., MIT, Stanford, Harvard, University of Michigan, Duke), and has been incorporated into the networking courses at Stanford, MIT, and NYU.

3 Future Research

I am excited to build systems that address the interdisciplinary challenges associated with future applications as they support richer, more personalized services and increased interactivity. I will describe several projects that relate to my recent research, and then discuss more speculative directions I am interested in pursuing.

3.1 Debugging and Monitoring Microservices

Many web applications like Uber and Netflix have adopted the "microservice" architecture, in which backends are composed of hundreds or thousands of small, loosely coupled services. This approach enables rapid development and testing of individual services. However, the proliferation of cross-service interactions creates significant problems for debugging and performance optimization.

As part of my future research, I would like to simplify these tasks by applying fine-grained data flow tracking to microservice-based applications. Existing tracing frameworks for distributed systems log coarse-grained events such as incoming client requests, particular system call invocations, or the execution of manually-specified lines of code. In contrast, I propose that a distributed application log *all* reads and writes to *all* program variables. By tracking this information, we can better understand the execution behavior of each microservice, and recreate a "global" view of the overall distributed system.

I have looked at similar problems with Cascade, but those assumed few server-side components. Microservice environments raise new scalability and heterogeneity questions. How can we quickly process data flows to support real-time monitoring? Can we prune logging or storage to improve efficiency, without sacrificing insights? How can we track provenance across services that use diverse data schemas and storage layers?

3.2 Smarter Edge Computing with Datacenter Sharding

Modern edge servers are not full-fledged participants in large-scale web services. Edge servers cache static content or host ephemeral computations, but *canonical application state*—the user information, service data, and service logic that constitute the heart of an application—only reside in datacenters. Thus, only datacenters can handle requests involving dynamic, personalized content, limiting the benefits of edge servers.

I propose that web services shard their canonical state across the wide area. A *datacenter shard* runs at the edge and hosts a complete replica of the application software stack, including application servers and persistent storage. In this way, shards can locally respond to client requests for dynamic content as datacenters do, without incurring costly wide-area RTTs. Importantly, shards *selectively* replicate an application's data objects to respond to a large fraction of client requests locally, while remaining practical.

Datacenter sharding introduces many challenges. Which data objects should be pushed to shards? What storage and compute resources do shards need to be effective? How can datacenters trust shards with sensitive data? What protocols can populate and update shards while preserving existing consistency guarantees (e.g., eventual, causal)? Answering these questions starts with identifying the state needed to respond to each client request. One approach is to instrument web frameworks and storage layer APIs to log the data objects (e.g., key/value entries, database rows) accessed by requests, but this may not work for all applications.

3.3 Enabling Applications to Leverage Accurate User Behavior Prediction

Web services commonly try to improve performance by executing tasks in anticipation of client requests. For example, CDNs are prepopulated with static content, while backend servers precompute responses and push resources to clients prior to their requests. The performance of these techniques depends heavily on the accuracy of user behavior prediction; inaccurate predictions can degrade performance and waste resources.

User devices and web services are tracking increasing amounts of behavioral data. However, applications are often not configured to reap all of the benefits that this rich data offers. As part of my future research, I plan to explore new design principles in which applications adjust execution using predictions about user behavior. For example, video players can use predictions of upcoming user seeks to start downloading later chunks; players today download chunks in order, responding to seeks only after they occur. Web servers could prioritize content in pages according to predictions of user interaction. Mobile operating systems could prioritize the transmission of cross-application data according to what users are predicted to do next.

Many practical challenges exist with this direction. What inputs are necessary for accurate prediction? How can applications handle inaccurate predictions? Can applications leverage rich network information (e.g., link capacity) to safely optimize predicted behavior? Where can data processing algorithms run to limit overhead but still provide useful insights? How should individual and aggregate predictions be combined?

3.4 Web Proxies for Encrypted Traffic

Remote dependency resolution (RDR) proxies load pages on behalf of clients to hide RTTs. These proxies must inspect cleartext objects to identify dependent resources, and are thus hindered by encrypted HTTPS content. Lodestar [7] overcomes this issue by having each origin run a proxy for its own HTTPS content. Yet a client could ideally use a single, origin-agnostic proxy that identifies all of a page's objects without learning their cleartext data. Existing cryptographic techniques let proxies search for prespecified strings in encrypted data, or decrypt entire traffic streams with endpoint permission. However, these approaches are insufficient for proxies to perform encrypted computation over ciphertext objects to discover dependencies.

I am interested in developing proxies that can perform RDR directly on encrypted objects. One approach is to use functional encryption, which allows an untrusted party to compute over encrypted data, generating cleartext outputs without learning anything about the ciphertext input. However, functional encryption is slow for RDR. Proxies may also be able to use trusted execution environments like SGX for secure parsing. I believe that my experience with RDR proxies will help find ways to reduce the encrypted computation required to proxy pages, and I am excited to collaborate with cryptographers to explore this direction.

References

[1] H. Mao, **R. Netravali**, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*, 2017.

[2] **R. Netravali**, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *USENIX NSDI*, 2016.

[3] **R. Netravali** and J. Mickens. Cascade: Using Data Flow Analysis For Speculative "What If?" Debugging of Web Applications. In Submission.

[4] **R. Netravali** and J. Mickens. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *USENIX NSDI*, 2018.

[5] **R. Netravali**, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Modern Web Pages. In *USENIX NSDI*, 2018.

[6] **R. Netravali**, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX Annual Technical Conference (ATC)*, 2015.

[7] **R. Netravali**, A. Sivaraman, J. Mickens, and H. Balakrishnan. Lodestar: Fast, Secure Page Loads Using Remote Dependency Resolution. In Submission.