Research Statement

John Vilk
jvilk@cs.umass.edu

The web browser is the most important application runtime today, encompassing all types of applications on practically every Internet-connected device. Browsers power complete office suites, media players, games, and augmented and virtual reality experiences, and they integrate with cameras, microphones, GPSes, and other sensors available on computing devices. Many apparently native mobile and desktop applications are secretly hybrid apps that contain a mix of native and browser code. History has shown that when new devices, sensors, and experiences appear on the market, the browser will evolve to support them.

Despite the browser's importance, developing web applications is exceedingly difficult. Web browsers organically evolved from a document viewer into a ubiquitous program runtime. The browser's scripting language for web designers, JavaScript, has grown into the only universally supported programming language in the browser. Unfortunately, JavaScript is notoriously difficult to write and debug. The browser's high-level and event-driven I/O interfaces make it easy to add simple interactions to webpages, but these same interfaces lead to nondeterministic bugs and performance issues in larger applications. These bugs are challenging for developers to reason about and fix.

My research vision is to revolutionize web development. Existing web development tools are based on conventional techniques developed in the 1970s that are ill-suited for the unique challenges posed by the browser. My research explores novel techniques that drive new *real-world* development tools aimed at helping developers overcome these challenges. My research addresses three fundamental questions related to web programming: *Can we make web programming easier? Can we make web applications easier to debug? Can we make web applications leaner and faster?*

My work, supported by a Facebook PhD Fellowship, has already had extensive real world impact. My research powers historical software that runs in the browser at the Internet Archive, is included in the ChakraCore JavaScript engine that ships with Windows 10, and has led to dozens of bug fixes in popular JavaScript libraries in wide use across the Internet.

Below, I discuss my current research that focuses on the client-side of web applications. I then outline future research directions, which extend the scope of my research to tackle problems that cross the client/server boundary.

Conventional Programming Language Support in the Browser

Can we bring conventional languages to the browser? The client-side of web applications must currently be written in JavaScript because that is the only programming language available in the browser. A few projects attempt to bring conventional languages like C/C++, Java, and Python to the browser on top of JavaScript. Ultimately, these efforts fall short because simply translating existing code into JavaScript is not enough.

The browser lacks practically all of the environment that code written in conventional languages expect, including processes, threads, a file system, and sockets. In short, directly implementing these resources on top of browser APIs is not generally possible because of an impedance mismatch between conventional languages and the browser. Browsers are not designed to support these resources. Developers must significantly rework or reimplement their code to cope with the limitations of the browser.

I led the design and development of Doppio [4] and Browsix [8], which combine to bridge the gap between conventional languages and the browser. **These systems make it possible to run unmodified code written in conventional languages in the browser.** Both of these systems are written completely in JavaScript. Doppio emulates POSIX-like resources including multiple threads, an extensible file system, and TCP sockets. Doppio's threads emulate preemptive multithreading on top of the browser's cooperative tasks, enabling code running with Doppio to issue blocking requests to its POSIX resources that interact with non-blocking browser APIs. Browsix extends Doppio with a kernel that provides centralized Unix-like resources, letting multi-process systems run in the browser.

Language implementations that target the browser can now use Doppio and Browsix to implement standard library and language features that require operating system support, enhancing their compatibility with existing code. I used Doppio to build DoppioJVM, a complete JVM implementation that runs unmodified JVM programs and languages completely in the browser. DoppioJVM powers a Java compiler and interactive games at CodeMoo.com from the University of Illinois, which teaches students how to code in Java. The Internet Archive uses Doppio to run historical software, including Windows 3.1 and the Oregon Trail, in the browser, which has already had over 31 million visitors. A Browsix-enhanced C/C++ to JavaScript compiler made it possible to bring a complete LaTeX development environment entirely into the browser with usable performance.[1] This work has indirectly impacted the evolution of browsers: Chrome developers explicitly cited the LaTeX demo as a reason to push an experimental browser feature into production.[2]

Time-Travel Debugging for Web Applications

Can we make web applications easier to debug? Web applications are difficult to debug because they are plagued with nondeterminism. Network requests can intermittently fail or return unexpected results. Browser events can race with one another. JavaScript code can race with the browser itself. Traditional stepping debuggers, which let developers place breakpoints and step forward through an execution, provide little help to developers trying to debug these issues. If a developer steps too far forward in the program, they may pass by the problematic line of code and need to restart the debugging process. Worse, the act of debugging itself can change the program schedule and prevent a bug from occurring because browser components execute concurrently with JavaScript.

Time-traveling debuggers offer the promise of removing much of this frustration. Using these systems, a developer only needs to record a problematic execution once; the developer can then freely time-travel to different points in the execution. Previous time-traveling debuggers for web applications are unable to support fast or precise time-travel because they treat the browser as a black box [1, 7]. These debuggers do not have access to internal browser state needed to checkpoint web applications mid-session, and thus are only able to time-travel by replaying from the beginning of a session. Black-box debuggers also assume that controlling JavaScript nondeterminism is sufficient to replay recorded executions, but the browser itself contains nondeterminism that these debuggers cannot control.

These problems are intractable if the browser is treated as a black box. However, I observe that the browser can be modified to expose *just enough additional state* to support time-travel. With this gray-box approach in mind, I led the design and development of ReJS [5], the first time-traveling debugger for web applications that supports precise and efficient time-travel. ReJS enhances a traditional stepping debugger with time-traveling operations, such as *stepping back* to the previous line, that would be infeasible to implement in a black box system. **ReJS faithfully and efficiently reproduces browser nondeterminism, letting developers debug backward and forward in time to isolate a bug's root cause.** ReJS also supports existing GUI debugging tools during time-travel because it keeps the browser's native GUI interfaces live and in sync with JavaScript execution.

ReJS extends the browser with non-intrusive *interrogative interfaces* that expose previously hidden runtime information. During an execution, ReJS uses these interfaces to log browser nondeterminism and take periodic snapshots of the application's state. ReJS imposes imperceptible tracing overhead, its application traces are small and portable, and, in the common case, stepping backwards only takes about a third of a second. Core parts of ReJS have been incorporated into Microsoft's ChakraCore JavaScript engine that ships with Windows 10 and form the basis of a new time-traveling debugger.

[1] Demos available at `https://doppiojvm.org/` and `http://browsertex.org/`

[2] The feature is `SharedArrayBuffer`: `https://goo.gl/x9bXqX`

Automatically Debugging Memory Leaks in Web Applications

Can we reduce the memory consumption of web applications? Browsers have an established reputation for consuming significant amounts of memory, and memory leaks in web applications only make matters worse. These leaks occur when the application references unneeded state, preventing the garbage collector (GC) from collecting it. Leaks degrade responsiveness by increasing GC frequency and overhead, and can even lead to browser tab crashes by exhausting available memory.

Despite the fact that memory leaks in web applications are a serious and pervasive problem, there are no automated tools that can find them. Existing leak detection techniques that work for C, C++, and Java are ineffective in the browser: leaks in web applications are fundamentally different from leaks in conventional applications. Developers are currently forced to manually inspect heap snapshots to locate objects that the application incorrectly retains. Unfortunately, these snapshots do not necessarily provide actionable information. They simultaneously provide too much information (every single object on the heap) and not enough information to actually debug these leaks (no connection to the code responsible for leaks). The result is that even expert developers are unable to find leaks: for example, a Google developer closed a Google Maps SDK bug report about a memory leak (with 99 stars and 51 comments) because it was "infeasible" to fix as they were "not really sure in how many places [it's] leaking." [3]

I built BLeak [3] (**B**rowser **Leak** debugger), the first system for automatically debugging memory leaks in web applications. BLeak leverages the following fact: over a single session, users repeatedly return to the same visual state. For example, Facebook users repeatedly return to the news feed and Gmail users repeatedly return to the inbox view. I observe that *these round trips can be viewed as an oracle to identify leaks*. Because visits to the same visual state should consume roughly the same amount of memory, sustained memory growth between visits is a strong indicator of a memory leak. BLeak builds directly on this observation to find memory leaks in web applications with high precision.

To use BLeak, a developer provides a short script (≈40 lines of code) to drive a web application in a loop that takes round trips through a specific visual state. **BLeak then proceeds automatically, identifying memory leaks, ranking them by their severity, and reporting their root cause in the source code.** On a corpus of production web applications, BLeak has a medium precision of 100% and precisely identifies the code responsible for nearly all of the leaks it finds (all but one). At least 77% of these leaks would not have been found by a conventional staleness-based approach. Fixing these leaks reduces heap growth by 94% on average, saving from 0.5 MB to 8 MB per return trip to the same visual state. Guided by BLeak, I identified and fixed over 50 memory leaks in popular libraries and applications including Airbnb, AngularJS, Google Analytics, and Google Maps SDK.

Future Research

My research so far has focused on the browser, but web applications typically consist of both browser and server components. For example, the Facebook web application consists of a client-side that runs in the browser and interacts with server-side microservices that power individual features like the news feed and chat. Correctness and performance bugs can cascade through these components, causing emergent behavior that is difficult to reason about. In the future, I plan to extend the scope of my research to tackle problems that cross the client/server boundary. I outline a few specific directions below.

Debugging across the wire

Standard debuggers are limited to examining a single program at a time, but web applications typically have separate client and server components. Complex bugs can occur when components interact

[3] `https://issuetracker.google.com/issues/35821412`

in unintended ways, fail unexpectedly, or make invalid assumptions about other components in the system. Developers commonly analyze log files from individual components to reason about whole system behavior, which is a manual and error-prone process. As a result, bugs involving multiple application components are challenging to locate and diagnose.

I plan to develop novel debugging techniques to help developers locate and diagnose bugs that cross the client/server boundary. A debugger that functions *across the wire* would let developers debug an entire web application at once and test hypotheses about whole-system behavior. For example, these debuggers could trace the *provenance* of specific events in the web application, and lead developers from problematic client-side behavior to code running on a server (or vice versa). Integrating low-overhead tracing and time-travel debugging would enable developers to use recorded traces to diagnose bugs in production systems after incidents occur. Such an approach poses consistency and scaling challenges that might be intractable in a general purpose distributed system, e.g., for whole-system snapshots, but can be made tractable in the web application setting by focusing on per-session state.

Performance debugging across the wire

Web application performance is vital because users quickly abandon unresponsive sites out of frustration, which can lead to a loss of revenue [6]. Performance is also a first-class concern in emerging augmented and virtual reality applications on the web: inconsistent frame rates induce motion sickness which *literally* makes users physically ill. Developers focus considerable effort on optimizing their web applications to operate quickly, smoothly, and responsively, but existing profilers do not provide enough information to guide developers to optimization opportunities.

Causal profiling is a new technique for predicting the effect of optimizing specific parts of a program on a throughput- or latency-based metric of interest [2]. A causal profiler conducts periodic performance experiments at runtime to directly measure the impact of a potential optimization. For each experiment, a causal profiler applies a virtual speedup to a region of code, then measures the impact on performance. Virtual speedups emulate the effect of a real speedup by delaying concurrently executing tasks, such as other threads, every time the sped-up code executes.

I have already implemented a prototype causal profiler for the client-side of web applications that operates on the browser's event-based concurrency primitives. I plan to extend this work to function across the client/server boundary. A causal profiler that functions *across the wire* will quantify improvements in client-side metrics like latency or frame rate resulting from server-side optimizations (and vice versa). Creating such a profiler entails revisiting core causal profiling concepts to function in a distributed fashion so as to decentralize performance experiments and enable virtual speedup delays to propagate across components in the system. A causal profiler that functions across the wire would be able to locate, precisely quantify, and rank optimization opportunities in both the client and server, letting the developer focus on optimizations that actually matter.

References

[1] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Symposium on User Interface Software and Technology*, pages 473–484, 2013.

[2] C. Curtsinger and E. D. Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.

[3] **J. Vilk** and E. D. Berger. BLeak: Automatically Debugging Memory Leaks in Web Applications. In submission to PLDI 2018.

[4] **J. Vilk** and E. D. Berger. Doppio: Breaking the Browser Language Barrier. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 508–518, 2014.

[5] **J. Vilk**, J. Mickens, and M. Marron. A Gray Box Approach For High-Fidelity, High-Speed Time-Travel Debugging. Technical Report MSR-TR-2016-7, Microsoft Research, June 2016.

[6] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *Internet Measurement Conference*, 2012.

[7] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for Javascript applications. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, pages 159–174, 2010.

[8] B. Powers, **J. Vilk**, and E. D. Berger. Browsix: Bridging the gap between unix and the browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017*, pages 253–266, 2017.