Live Monitoring: Using Adaptive Instrumentation and Analysis to Debug and Maintain Web Applications

Emre Kıcıman and Helen J. Wang
Microsoft Research
{*emrek, helenw*}*@microsoft.com*

Abstract

AJAX-based web applications are enabling the next generation of rich, client-side web applications, but today's web application developers do not have the end-to-end visibility required to effectively build and maintain a reliable system. We argue that a new capability of the web application environment—the ability for a system to automatically create and serve different versions of an application to each user—can be exploited for adaptive, cross-user monitoring of the behavior of web applications on end-user desktops. In this paper, we propose a live monitoring framework for building a new class of development and maintenance techniques that use a continuous loop of automatic, adaptive application rewriting, observation and analysis. We outline two such adaptive techniques for localizing data corruption bugs and automatically placing function result caching. The live monitoring framework requires only minor changes to web application servers, no changes to application code and no modifications to existing browsers.

1 Introduction

Over the last several years, AJAX (Asynchronous JavaScript and XML) programming techniques have enabled a new generation of popular web-based applications, marking a paradigm shift in web service development and provisioning [11]. Unlike traditional web services, these new *web applications* combine the data preservation and integrity, storage capacity and computational power of data center(s) with a rich client-side experience, implemented as a JavaScript program shipped on-demand to users' web browsers[1]. This combination provides a compelling way to build new applications while moving the burden of managing an application's reliability from end-users to the application's own developers and operators.

Unfortunately, today's web application developers and operators do not have the end-to-end visibility they need to effectively build and maintain a dependable system. Unlike traditional web services, running exclusively in controlled, server-side environments, a web application depends on many components outside the developer's control, including the client-side JavaScript engine and libraries and the third-party back-end web services used by *mash-up* applications—web applications that combine functionality from multiple back-end web services. Of course, web application developers must also contend with the traditional bugs that occur when writing any large, complex piece of software, including logic errors, memory leaks and performance problems. When the inevitable problem does occur, the web application developer's lack of visibility into the heterogeneous client environments and the dynamic behavior of third-party services can make reproducing and debugging the problem practically impossible.

To address these challenges, we propose a *live monitoring* framework that exploits a new capability of the web application environment, *instant redeployability*: Each time any client runs a web application, the developers and operators of the application can automatically provide the client a new, different version of the application. Our live monitoring framework (1) exploits this capability to enable dynamic and adaptive instrumentation strategies; and (2) integrates the resultant on-line observations of an application's end-to-end behavior into the development and operations process.

Live monitoring enables a new class of techniques that use a continuous loop of automatic application rewriting, observation and analysis to improve the development and maintenance of web applications. Policy-based, automatic rewriting of application code provides the necessary visibility into end-to-end application behavior, and collecting observations on-line from live end-user desk-

[1] We make a distinction between a web service and a web application. The former includes only server-side components, while the latter also includes a significant client-side JavaScript component

Op	Performance (ms)	
	IE 7	**Firefox 1.5**
Array.join	35	120
+	5100	120

Table 1: The performance of two simple methods for concatenating 10k strings varies across browsers.

tops provides visibility into the real problems affecting clients. Distributing and sampling instrumentation across the many users of a web application provides a low-overhead instrumentation platform. Finally, using already-collected information to adapt instrumentation on-line enables efficient drill-down with specialized diagnosis techniques as problems occur.

2 Reliable Web Applications

The web application environment presents many of the same development and operations challenges that confront any cross-platform, distributed system. In this environment, however, there are also opportunities for a new approach to addressing these challenges.

Challenges

The root challenge to building and maintaining a reliable client-side web application is a lack of visibility into the end-to-end behavior of the program, brought about by the fact that execution of the web application is now split across multiple environments, including uncontrolled client-side and third-party environments and exacerbated by their heterogeneity and dynamics.

Non-standard Execution Environments: While the core JavaScript language is standardized as ECMAScript [7], most pieces of a JavaScript environment are not. The result is that applications have to frequently work-around subtle and not-so-subtle cross-browser incompatibilities. As a clear example, sending an XMLRPC request requires calling an ActiveX object in IE6, but a native JavaScript object in Firefox. More subtle are issues such as event propagation: *e.g.*, given multiple registered event handlers for a mouse click, in what order are they called? Moreover, even the standardized pieces of JavaScript can have implementation differences that cause serious performance problems (see Table 1 for examples of performance variance across browsers.)

Third-Party Dependencies: All web applications have dependencies on the reliability of back-end web services. While these back-end services strive to maintain high-availability, they can and do fail. Moreover, even regular updates, such as bug fixes and feature enhancements

App	JS (bytes)	JS (LOC)
Live Maps	1MB	54K
Google Maps	200KB	20K
HousingMaps	213KB	19K
Amazon Book Reader	390KB	16K
CNN.com	137KB	5K

Table 2: Numbers on the amount of client-side code in a few major web applications, measured in bytes and lines of code (LOC)

can break dependent applications. Anecdotally, such breaking upgrades do occur: Live.com updated their beta gadget API, breaking dependent developers code [13]; and, more recently, the popular social bookmark website, del.icio.us, moved the URLs pointing to some of their public data streams, breaking dependent applications [3].

Software Complexity: Of course, JavaScript also suffers from the traditional challenges of writing any non-trivial program[2]. While JavaScript programs were once only simple scripts containing a few lines of code, they have grown dramatically to the point where the client-side code of cutting-edge web applications easily exceed 10k lines, as shown in Table 2. The result is that web applications suffer from the same kinds of bugs as traditional programs, including memory leaks, logic bugs, race conditions, and performance problems.

The difficulties caused by heterogeneous execution environments and dynamic third-party behavior, as well as the challenge of writing correct software can certainly be improved through more complete standardization, better web service management and careful software engineering. But, we would argue that, at a minimum, software bugs and human error will continue to give all of these challenges a long life frustrating web application developers.

Opportunities

While the above challenges are faced by most any cross-platform distributed systems, two technical features of web applications provide an opportunity for building new kinds of tools to deal with these problems:

Instant Deployability: Web applications are deployed and updated by modifying the code stored on a central web server. Modulo caching policies, clients download a fresh copy of the application each time they run it, enabling instant deployability of updates. We take advan-

[2]Coding in JavaScript today is also made more difficult by a lack of compile-time errors and warnings, static type checking, and private scoping. We do not consider these problems fundamental, however, as current and upcoming tools, such as Google's WebToolkit are remedying these issues [8].

tage of this capability to serve different versions of a web application (*e.g.*, with varying instrumentation) over time and across users.

Dynamic extensions: During their execution, JavaScript-based web applications can dynamically load and run new scripts, allowing late-binding of functionality based on current requirements. We use this to download specialized fault diagnosis routines when a web application encounters a problem.

3 Live Monitoring

The goal of live monitoring techniques is to improve developer and operator visibility into the end-to-end behavior of web applications by enabling automatic, adaptive analysis of application behavior on real end-user desktops. At the core of a live monitoring technique is a simple process:

1. Use automatic program rewriting together with instant redeployability to serve differently instrumented versions of applications over time and across users.

2. Continually gather observations of the on-line, end-to-end behavior of applications running under real workload on many end-user's desktops.

3. As observations of application behavior are gathered and analyzed, use the results to guide the adaptation of the dynamic instrumentation policy.

4. In special cases, use the client's ability to dynamically load scripts to enable just-in-time fault diagnosis handlers, tailored based on previously gleaned information about the specific encountered symptoms.

Our framework for live monitoring, shown in Figure 1, divides this process across several key components. The **Transformer** is responsible for rewriting the JavaScript application as it is sent from the web application's servers to the end-user's desktop. The transformer contains both generic code, such as the JavaScript and HTML parsers reusable across many live monitoring techniques, and technique-specific rewriting rules. These rules are expressed in two steps: the first step searches for target code-points matching some rule-specific filter, such as "all function call expressions" or "all variable declarations"; and the second step applies an arbitrary transformation on a target code-point by modifying the abstract syntax tree of the JavaScript program. Each rewriting rule exposes a set of discrete knobs for controlling the rewriting of target code points. For example, a rule that

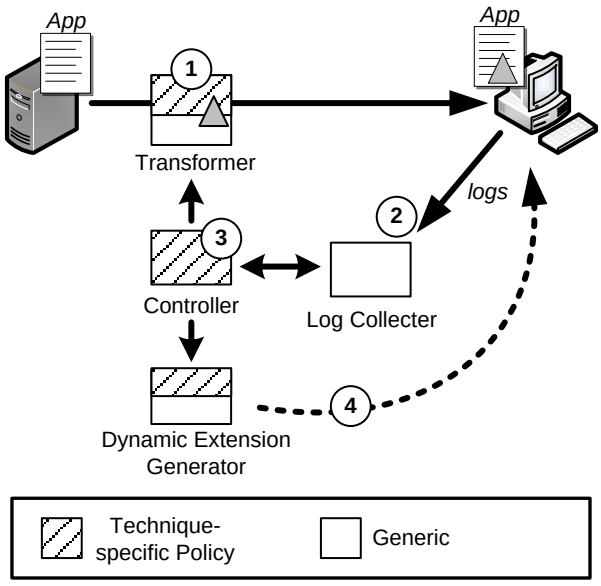

Figure 1: Live Monitoring Framework.

adds performance profiling to function calls might expose an on/off knob for each function that could be profiled.

The **Controller** component is responsible for the core of the technique-specific adaptation algorithm, analyzing the collected observations of application behavior and using the results of the analysis to modify the knobs exposed by the rewriting rules in the Transformer. The **Log Collector** is a simple component, responsible for gathering observations returned by rewritten programs; and the **Dynamic Extension Generator** creates special-purpose fault diagnosis handlers, based on the application's request and configuration input from the Controller.

While some parts of this process are generic and reusable across techniques, the rest—what we call a live monitoring policy—is specific to each live monitoring technique. This policy includes the rewriting rules in the Transformer, the analysis policy in the Controller responsible for analyzing logs and modifying the knobs of the rewriting rules, and the dynamic extension generator.

4 Live Monitoring Policies

When developing a new policy to address a debugging or maintenance challenge, we consider several questions:

What are the appropriate rewriting rules? The first consideration when building a monitoring policy is what observations of application behavior need to be captured, and how a program can be modified to efficiently capture it. In particular, we ask what instrumentation is statically

written into the code, and what functionality will be dynamically determined and downloaded as needed from the Dynamic Extension Generator.

How does the rewriting adapt over time? A second consideration is which code points in a program should initially be rewritten, and how this choice changes over time as we gather more observations of behavior. The policy should also consider whether a multi-stage approach is appropriate, where completely different rewriting rules are applied to gather different kinds of information over time.

How does the policy spread instrumentation across users? A third axis of consideration is how a policy can distribute instrumentation across many users (*e.g.*, via sampling) and re-aggregate that information to reason about the program's overall behavior.

How do the developers and operators interact and use live monitoring policies? The final question when designing a policy is how people will use it. Some policies may be completely automated and continuously running, whereas other live monitoring policies may only run occasionally and on the explicit request of a developer. In particular, if the policy's application rewriting might affect the semantics of the program then human interaction is likely necessary.

We have built a prototype of our live monitoring framework, implemented several policies for debugging errors, drilling-down into performance problems, and analyzing runtime behavior to detect potentially correct cache optimizations and are exploring answers to these questions. The rest of this section describes two policies that use different styles of adaptation to address different problems. In the first example, a single rewriting rule is applied to different points in the code as we drill-down into data structure corruptions. The second example uses different rewriting rules over time, and decides where to place each rewriting rule based on observations gathered from previous application runs.

Locating Data Structure Corruption Bugs

While it can be very difficult to reproduce the steps to triggering bugs in a controlled, development environment, real users will run into the same problems again and again in a real, deployed application. We would like to capture the relevant error information and debug problems in real conditions, but adding all the necessary debugging infrastructure to the entire program can have too high an overhead. The solution is to adaptively enable the debugging infrastructure only when and where in the code it is needed.

Corruption of in-memory data structures is a clear sign of a bug in an application, and can easily lead to serious problems in the application's behavior. A straightforward method for detecting data structure inconsistencies is to use consistency checks at appropriate locations to ensure that data structures are not corrupt. A consistency check is a small piece of data-structure-specific code that tests for some invariant. *E.g.*, a doubly-linked list data structure might be inspected for unmatched forward and backward references. While today these checks are commonly written manually, there has been recent work on automatically inferring such checks [6].

When a consistency check fails, we might suspect that a bug exists somewhere in the executed code after the last successful consistency check[3]. If we execute these consistency checks infrequently, we will not have narrowed down the possible locations of a bug. On the other hand, if we execute these checks too frequently, we can easily cause a prohibitive performance overhead, as well as introduce false positives if we check a data structure while it is being modified.

Using live monitoring, we can build an adaptive policy that adds and removes consistency checks to balance the need for localizing data structures with the desire to avoid excessive overhead. Initially, the policy inserts consistency checks only at the beginning and end of stand-alone script blocks and event handlers (essentially, all the entry and exit points for the execution of a JavaScript application). Assuming that any data structure that is corrupted during the execution of a script block or event handler will remain corrupted at the end of the block's execution, we have a high confidence of detecting corruptions as they are caused by real workloads.

As these consistency checks notice data structure corruptions, the policy adds additional consistency checks in the suspect code path to "drill-down" and help localize the problem. As clients download and execute fresh copies of the application and run into the same data structure consistency problems, they will report in more detail on any problems they encounter in this suspect code path, and our adaptive policy can then drill-down again, as well as remove any checks that are now believed to be superfluous.

Several simple extensions can make this example policy more powerful. For example, performance overhead can be reduced at the expense of fidelity by randomly sampling data structure consistency across many clients. Also, if the policy finds a function that only intermittently corrupts a data structure, we can explore the program's state in more detail with an additional rewriting rule to capture the function's input arguments and other key state arguments and other state to help the developer narrow down the cause of a problem.

[3] JavaScript programs are executed within a single-thread, avoiding the possibility of a separate thread having corrupted the data structure.

Identifying Promising Cache Placements

Even simple features of web applications are often cut because of performance problems, and the poor performance of overly ambitious AJAX applications is one of the primary complaints of end-users. Some of the blame lies with JavaScript's nature as a scripting language not designed for building large applications: given a lack of access scoping and the ability to dynamically load arbitrary code, the scripting engine often cannot safely apply even simple optimizations, such as caching variable dereferences and in-lining functions.

With live monitoring, however, we can use a multi-stage instrumentation policy to detect possibly valid optimizations and evaluate the potential benefit of applying the optimization. Let us consider a simple optimization strategy: the insertion of function result caching. For this optimization strategy to be correct, the function being cached must (1) return a value that is deterministic given only the function inputs and (2) have no side-effects. We monitor the dynamic behavior of the application to cull functions that empirically do not meet the first criteria. Then, we rely on a human developer to understand the semantics of the remaining functions to double-check the remaining functions for determinism and side-effects. Finally, we use another stage of instrumentation to check whether the benefit of caching outweighs the cost.

The first stage of such a policy injects test predicates to help identify when function caching is valid. To accomplish this, the rewriting rule essentially inserts a cache, but continues to call the original function and check its return value against any previously cached results. If any client, across all the real workload of an application, reports that a cache value did not match the function's actual return value, we know that function is not safe for optimization and remove that code location from consideration. After gathering many observations over a sufficient variety and number of user workloads, we provide a list of potentially cache-able functions to the developer of the application and ask them to use their knowledge of the function's semantics to determine whether it might have any side-effects or unseen non-determinism. The advantage of this first stage of monitoring is that reviewing a potentially short list of possibly valid cache-able code points should be easier than inspecting all the functions for potential cache optimization.

In the second stage of our policy, we use automatic rewriting to cache the results of functions that the developer deemed to be free of side-effects. To test the cost and benefit of each function's caching, we distribute two versions of the application: one with the optimization and one without, where both versions have performance instrumentation added. Over time, we compare our observations of the two versions and determine when and where the optimization has benefit. For example, some might improve performance on one browser but not another. Other caches might have a benefit when network latency between the user and the central service is high, but not otherwise. Regardless, testing optimizations in the context of a real-world deployment, as opposed to testing only in a controlled pre-deployment environment, allows us to evaluate performance improvement while avoiding any potential systematic biases of test workloads or differences between real-world and test environments.

5 Related Work

Several previous projects have worked on improved monitoring techniques for web services and other distributed systems [2, 1], but to our knowledge, live monitoring is the first to extend developer's visibility into web application behavior on the end-user's desktop. Others, including Tucek *et al* [15], note that moving debugging capability to the end-user's desktop benefits from leveraging information easily available only at the moment of failure—we strongly agree. In [9] Liblit *et al* present an algorithm for isolating bugs in code by using randomly sampled predicates of program behavior from a large user base. We believe that the adaptive instrumentation of live monitoring can improve on such algorithms by enabling the use of active learning techniques [5] that use global information about encountered problems to dynamically control predicate sampling. Perhaps the closest in spirit to our work is ParaDyn [10], which uses dynamic, adaptive instrumentation to find performance bottlenecks in parallel computing applications.

6 Challenges and Implications

In summary, we have presented live monitoring, a framework for improving developers' end-to-end visibility into web application behavior through a continuous, adaptive loop of instrumentation, observation and analysis. As examples, we have shown how live monitoring can be used to localize bugs and analyze runtime behavior to detect and evaluate optimization opportunities.

We still face open challenges as we look to building a practical and deployable live monitoring system, such as the privacy issues of added instrumentation. While we believe that the browser's sandbox on web applications, together with the explicit trust users already give web services to store application-specific personal information (e-mails, purchasing habits, etc) greatly reduces the potential privacy concerns of extra instrumentation, there may be corner cases where live monitoring would pose a risk. Another challenge is to maintain the

predictability—predictable behavior and performance—of web applications as we dynamically adapt our instrumentation.

If successful, however, we believe the implications of instant redeployability may go beyond monitoring and also open the door to adaptive recovery techniques, including variations of failure-oblivious computing and Rx techniques [14, 12]. In these cases, the detection of a failure and the discovery of an appropriate mitigation technique in one user's execution of an application could be immediately applied to help other users, before they experience problems. At the moment, web applications are the most widespread platforms that have the capability for instantly redeployment. In the future, however, automatic update mechanisms and other centralized software management tools [4] might enable instant redeployability in broader domains.

References

[1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP 2003*.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of OSDI 2004*.

[3] A. Bosworth. How To Provide a Web API. `http://www.sourcelabs.com/blogs/ajb/2006/08/how_to_provide_a_web_api.html`, Aug 2006.

[4] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M.S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of NSDI 2005*.

[5] D.A. Cohn, Z. Ghahramani, and M.I. Jordan. Active Learning with Statistical Models. *Journal of Artificial Intelligence Research*, 4, 1996.

[6] B. Demsky, M.D. Ernst, P.J. Guo, S. McCamant, J.H. Perkins, and M. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of ISSTA 2006*.

[7] ECMA. ECMAScript Language Specification 3rd Ed. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec 1999.

[8] Google. Google web toolkit. `http://code.google.com/webtoolkit/`.

[9] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of PLDI 2005*.

[10] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11), Nov 1995.

[11] T. O'Reilly. What is Web 2.0. `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web20.html`, Sep 2005.

[12] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failure. In *Proceedings of SOSP 2005*.

[13] S. Rider. Recent changes that may break your gadgets. `http://microsoftgadgets.com/forums/1438/ShowPost.aspx`, Nov 2005.

[14] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and Jr. W.S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of OSDI 2004*.

[15] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Automatic On-line Failure Diagnosis at the End-User Site. In *Proceedings of HotDep 2006*.