AjaxScope: A Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications

EMRE KICIMAN and BENJAMIN LIVSHITS
Microsoft Research

The rise of the software-as-a-service paradigm has led to the development of a new breed of sophisticated, interactive applications often called Web 2.0. While Web applications have become larger and more complex, Web application developers today have little visibility into the end-to-end behavior of their systems. This article presents AjaxScope, a dynamic instrumentation platform that enables cross-user monitoring and just-in-time control of Web application behavior on end-user desktops. AjaxScope is a proxy that performs on-the-fly parsing and instrumentation of JavaScript code as it is sent to users' browsers. AjaxScope provides facilities for distributed and adaptive instrumentation in order to reduce the client-side overhead, while giving fine-grained visibility into the code-level behavior of Web applications. We present a variety of policies demonstrating the power of AjaxScope, ranging from simple error reporting and performance profiling to more complex memory leak detection and optimization analyses. We also apply our prototype to analyze the behavior of over 90 Web 2.0 applications and sites that use significant amounts of JavaScript.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Reliability, Performance, Measurement, Management, Languages

Additional Key Words and Phrases: Applications, software monitoring, software instrumentation

ACM Reference Format:
Kıcıman, E. and Livshits, B. 2010. AjaxScope: A platform for remotely monitoring the client-side behavior of Web 2.0 applications. ACM Trans. Web 4, 4, Article 13 (September 2010), 52 pages. DOI = 10.1145/1841909.1841910. http://doi.acm.org/10.1145/1841909.1841910.

1. INTRODUCTION

In the last several years, there has been a sea change in the way software is developed, deployed, and maintained. Much of this has been the result of

Authors' addresses: E. Kıcıman and B. Livshits, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: {emrek, livshits}@microsoft.com.

a rise of software-as-a-service paradigm as opposed to traditional shrink-wrap software. These changes have lead to an inherently more dynamic and fluid approach to software distribution, where users benefit from bug fixes and security updates instantly and without hassle. This fluidity also creates opportunities for software monitoring. Indeed, additional monitoring code can be seamlessly injected into the running software without the user's awareness.

Nowhere has this change in the software deployment model been more prominent than in a new generation of interactive and powerful Web applications. Sometimes referred to as Web 2.0, applications such as Yahoo! Mail and Google Maps have enjoyed wide adoption and are highly visible success stories. In contrast to traditional Web applications that perform the majority of their computation on the server, Web 2.0 applications include a significant client-side JavaScript component. Widely used applications consist of over 50,000 lines of JavaScript code executing in the user's browser. Based on AJAX (Asynchronous JavaScript and XML), these Web applications use dynamically downloaded JavaScript programs to combine a rich client-side experience with the storage capacity, computational power, and reliability of sophisticated data centers.

However, as Web applications grow larger and more complex, their dependability is challenged by many of the same issues that plague any large, cross-platform distributed system that crosses administrative boundaries. There are subtle and not-so-subtle incompatibilities in browser execution environments, unpredictable workloads, software bugs, dependencies on third-party Web services, and—perhaps most importantly—a lack of end-to-end visibility into the remote execution of the client-side code. Without visibility into client-side behavior, developers have to resort to explicit user feedback and attempts to reproduce user problems.

This article presents AjaxScope, a platform for instrumenting and remotely monitoring the client-side execution of Web applications within users' browsers. Our goal is to enable practical, flexible, fine-grained monitoring of Web application behavior across the many users of today's large Web applications. While coarse solutions to track page hit statistics or to optimize Web sites for conversion exist,[1] our primary focus is on enabling monitoring and analysis of program behavior at the source code level to improve developers' visibility into the correctness and performance problems being encountered by end-users. Our approach focuses on runtime instrumentation; a popular alternative is static analysis. However, for large applications, static analysis of JavaScript has been tricky because of the highly dynamic nature of this language [Yue and Wang 2009] and because JavaScript programs are often incomplete, making whole-program analysis difficult [Guarnieri and Livshits 2009; Chugh et al. 2009; Guarnieri and Livshits 2010].

To achieve this goal, we take advantage of a new capability of the Web application environment, *instant redeployability*: the ability to dynamically serve new, different versions of code each time any user runs a Web application. Indeed, because caching headers are controlled by the server, we can force

[1]http://www.omniture.com is one such solution.

refreshes every time an application is visited. We use this ability to dynamically provide differently instrumented code per user and per execution of an application.

Instant redeployability allows us to explore two novel instrumentation concepts, *adaptive instrumentation*, where instrumentation is dynamically added or removed from a program as its real-world behavior is observed across users; and *distributed tests*, where we distribute instrumentation and runtime analyses across many users' execution of an application, such that no single user experiences the overhead of heavyweight instrumentation. A combination of these techniques allows us to take many brute-force, runtime monitoring policies that would normally impose a prohibitively high overhead, and instead spread the overhead across users and time so that no single execution suffers too high an overhead. In addition, instant redeployability enables comparative evaluation of optimizations, bug fixes, and other code modifications.

To demonstrate these concepts, we built AjaxScope, a prototype proxy that rewrites JavaScript-based Web applications on the fly as they are being sent to a user's browser. AjaxScope provides a flexible, policy-based platform for injecting arbitrary instrumentation code to monitor and report on the dynamic runtime behavior of Web applications, including their runtime errors, performance, function call graphs, application state, and other information accessible from within a Web browser's JavaScript sandbox. Because our prototype can parse and rewrite standard JavaScript code,[2] it does not require changes to the server-side infrastructure of Web applications, nor does it require any extra plug-ins or extensions on the client browser. While we built our prototype to rewrite JavaScript code, our techniques may be extended to other forms of client-executable code, such as Flash or Silverlight. In addition to the client-side proxy, we have also implemented an AjaxScope plug-in for the IIS Web server, which provides sever-side JavaScript rewriting before the code is shipped to the user. The server-side approach may be used to collect fine-grained performance statistics for JavaScript code in a browser-independent manner.

To evaluate the flexibility and efficacy of AjaxScope, we use it to implement a range of developer-oriented monitoring policies, including runtime error reporting, drill-down performance profiling, optimization-related policies, a distributed memory leak checker, and a policy to search for and evaluate potential function cache placements. In the course of our experiments, we have applied these policies to 90 Web sites that use JavaScript.

1.1 Contributions

This article makes the following contributions.

—We demonstrate how instant redeployability of applications can provide a flexible platform for monitoring, debugging, and profiling of service-oriented applications.

[2]A public release of our prototype proxy, extensible via plug-in instrumentation policies, is available at http://research.microsoft.com/projects/ajaxview/.

—For Web 2.0 applications, we show how such a monitoring platform can be implemented using dynamic rewriting of client-side JavaScript code.

—We present two instrumentation techniques, adaptive instrumentation and distributed tests, and show that these techniques can dramatically reduce the per-user overhead of otherwise prohibitively expensive policies in practice. Additionally, we demonstrate how our platform can be used to enable on-line comparative evaluations of optimizations and other code changes.

—We evaluate the AjaxScope platform by implementing a variety of instrumentation policies and applying them to 90 Web applications and sites containing JavaScript code. Our experiments qualitatively demonstrate the flexibility and expressiveness of our platform and quantitatively evaluate the overhead of instrumentation and its reduction through distribution and adaptation.

—We implement and present several additional uses of AjaxScope, including MEDic, a control and visualization tool for Web application instrumentation to enable on-line debugging; and Doloto, a code splitter that optimizes Web application download time based on analysis of its runtime behavior.

1.2 Article Organization

The rest of the article is organized as follows. First, we give an overview of the challenges and opportunities that exist in Web application monitoring. Then, in Section 3, we describe the architecture of AjaxScope, together with example policies and design decisions. We present our implementation, as well as our experimental setup and microbenchmarks of our prototype's performance in Section 4. Section 5 and Section 6 describe adaptive instrumentation and distributed tests, using drill-down performance profiling and memory leak detection as examples, respectively. Section 7 discusses comparative evaluation, or A/B testing, and applies it to dynamically evaluate the benefits of cache placement choices. Sections 8 and 9 present two tools we have built atop AjaxScope: The first is MEDic, and consists of a library and manual control and visualization tool for instrumented Web applications. The second is Doloto, a code splitter that rewrites Web application code layout to optimize for download time. Section 10 outlines several additional, potential uses of AjaxScope's instrumentation capabilities. We discuss implications for Web application development and operations in Section 11. Finally, Sections 12 and 13 present related work and our conclusions.

2. OVERVIEW

Modern Web 2.0 applications share many of the development challenges of any complex software system. But the Web application environment also provides a number of key opportunities to simplify the development of monitoring and program analysis tools. The rest of this section details these challenges and opportunities, and presents concrete examples of monitoring policies demonstrating the range of possible capabilities.

Table I. Performance of Simple JavaScript Operations Varies Across Commonly Used Browsers. Time is Shown in Msec to Execute 10k Operations

Browser	**Version**	**Array.sort()**	**Array.join()**	**String +**
Internet Explorer	6.0	823	38	4820
Internet Explorer	7.0	833	34	4870
Opera	9.1	128	16	6
FireFox	1.5	261	124	142
FireFox	2.0	218	120	116

2.1 Core Challenges

The core challenge to building and maintaining a reliable client-side Web application is a lack of visibility into its end-to-end behavior across multiple environments and administrative domains. As described in the following, this lack of visibility is exacerbated by uncontrolled client-side and third-party environment dependencies and their heterogeneity and dynamics.

Nonstandard Execution Environments. While the core JavaScript language is standardized as ECMAScript [ECMA 1999], runtime JavaScript execution environments differ significantly. As a result, applications have to frequently work around subtle and not-so-subtle cross-browser incompatibilities. As a clear example, sending an XML-RPC request involves calling an ActiveX object in Internet Explorer 6, as opposed to a native JavaScript object in Mozilla FireFox. Other, more subtle issues include significant cross-browser differences in event propagation models. For example, given multiple event handlers registered for the same event, in what order are they executed? Moreover, even the standardized pieces of JavaScript can have implementation differences that cause serious variations in performance; see Table I for examples.

Third-Party Dependencies. All Web applications have dependencies on the reliability of back-end Web services. And though they strive to maintain high availability, these back-end services can and do fail. However, even regular updates, such as bug fixes and feature enhancements, can easily break dependent applications. Anecdotally, such breaking upgrades do occur: `live.com` updated their beta gadget API, breaking dependent developers code [Rider 2005]; and, more recently, the popular social bookmark Web site, `del.icio.us`, moved the URLs pointing to some of their public data streams, breaking dependent applications [Bosworth 2006].

Traditional Challenges. Where JavaScript programs used to be 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 tens of thousands of lines of code (see our selected benchmarks in Section 4.3). 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. Worse, JavaScript does not provide modularity constructs found in Java and C#, such as namespaces, which makes large applications even more challenging to develop because of the lack of isolation between different portions of the code.

2.2 Key Opportunities

While the challenges of developing and maintaining a reliable Web application are similar to traditional software challenges, there are also key opportunities in the context of rich-client Web applications, as detailed here.

Instant redeployment. In contrast to traditional desktop software, changes can be made immediately to Web 2.0 applications. While browsers and proxies do cache Web content, including JavaScript code, they do so in accordance with the cache control settings provided by a Web server. Often, top-level Web documents (i.e., HTML files) are given a short time-to-live, while the secondary—and often larger—Web objects (e.g., images, CSS files) are given infinite time-to-live. Updating the secondary Web objects involves assigning a new name and updating the reference in the primary document. Such techniques allow Web applications to achieve both the performance benefits of caching and the ability to provide fresh content to every client. AjaxScope takes advantage of this to perform on-the-fly, per-user JavaScript rewriting.

Adaptive and distributed instrumentation. Web 2.0 applications are inherently multi-user, which allows us to seamlessly distribute the instrumentation burden across a large user population. This enables the development of sophisticated instrumentation policies that would otherwise be prohibitively expensive in a single-user context. The possibility of adapting instrumentation over time enables further control over this process.

Large-scale workloads. In recent years, runtime program analysis has been demonstrated as an effective strategy for finding and preventing bugs in the field [Liblit et al. 2005; Martin et al. 2005]. Many Web 2.0 applications have an extensive user base, whose diverse activity can be observed in real time. As a result, a runtime analysis writer can leverage the high combined code coverage not typically available in a test context.

2.3 Categories of Instrumentation Policies

As a platform, AjaxScope enables a large number of exciting instrumentation policies.

Performance. Poor performance is one of the most commonly heard complaints about the current generation of AJAX applications [Breen 2007]. AjaxScope enables the development of policies ranging from general function-level performance profiling (Section 5.2) to timing specific parts of the application, such as initial page loading or the network latency of asynchronous AJAX calls. Section 9 talks about DOLOTO, a tool built on top of AjaxScope that does code splitting for optimization.

Runtime analysis and debugging. AjaxScope provides a platform for implementing a range of runtime analyses, from finding simple bugs like infinite loops (Section 3.2.2) to complex proactive debugging policies such as memory leak detection (Section 6.2). Given the large number of users for the more

Table II. Policies Described Before and in Sections 5–7. Simple Policies Above the Separator Line such as Error Reporting and Infinite Loop Detection are neither Adaptive nor Distributed

Policy	Adaptive	Distributed	A/B Test
Client-side error reporting			
Infinite loop detection			
String concatenation detection			
Performance profiling	✓		
Memory leak detection		✓	
Finding caching opportunities	✓	✓	
Testing caching opportunities			✓

popular applications, an AjaxScope policy is likely to enjoy high runtime code coverage.

Usability evaluation. AjaxScope can help perform usability evaluation. Because JavaScript makes it easy to intercept UI events such as mouse movements and key strokes, user activity can be recorded, aggregated, and studied to produce more intuitive Web interfaces [Atterer et al. 2006]. While usability evaluation is not a focus of this article, we discuss some of the privacy and security implications in Section 11.

The policies we implement and describe in this article are summarized in Table II.

3. AJAXSCOPE DESIGN

Here we first present a high-level overview of the dynamic instrumentation process and how it fits into the Web application environment, followed in Section 3.2 with some simple examples of how instrumentation can be embedded into JavaScript code to gather useful information for the development and debugging process. Sections 3.3 and 3.4 summarize the structure of AjaxScope instrumentation policies and policy nodes.

3.1 Platform Overview

Figure 1 shows how an AjaxScope proxy fits into the existing Web application environment. Other than the insertion of the server-side proxy, AjaxScope does not require any changes to existing Web application code or servers, nor does it require any modification of JavaScript-enabled Web browsers. The Web application provides uninstrumented JavaScript code, which is intercepted and dynamically rewritten by the AjaxScope proxy according to a set of instrumentation policies. The instrumented application is then sent on to the user. Because of the distributed and adaptive features of instrumentation policies, each user requesting to download a Web application may receive a differently instrumented version of code.

The instrumentation code and the application's original code are executed together within the user's JavaScript sandbox. The instrumentation code generates log messages recording its observations and queues these messages in

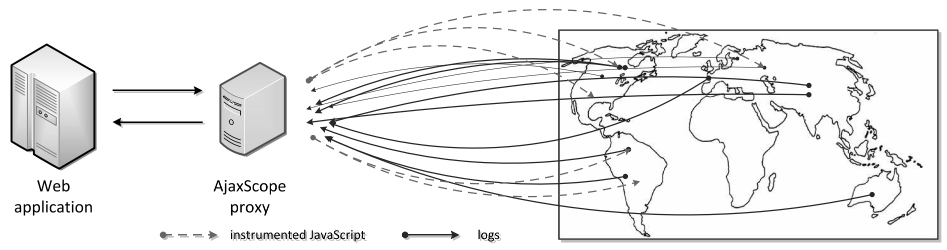


Fig. 1. Deployment of AjaxScope server-side proxy for a popular Web application lets developers monitor real-life client-side workloads.

memory. Periodically, the Web application collates and sends these log messages back to the AjaxScope proxy.

Remote procedure call responses and other data sent by the Web application are passed through the AjaxScope proxy unmodified, but other downloads of executable JavaScript code will be instrumented according to the same policies as the original application. JavaScript code that is dynamically generated on the client and executed via the `eval` construct is not instrumented by our proxy.

One option, left for future work, is to rewrite calls to `eval` to send dynamically generated scripts to the proxy for instrumentation before the script is executed, as explored elsewhere [Meyerovich and Livshits 2010]. While `eval` may be used for a variety of malicious purposes [Yue and Wang 2009], in our experience, well-behaved Web applications make rare use of dynamic code generation because of the performance penalty, reserving them for evaluating JSON (JavaScript Object Notation) or for dynamic code loading. AjaxScope recognizes the former pattern, avoiding rewriting. Rewriting well-formed non-JSON in the latter case in the proxy is also possible.

3.2 Example Policies

In the following, we describe three simple instrumentation schemes to illustrate how source-level automatic JavaScript instrumentation works. The purpose of these examples is to demonstrate the flexibility of instrumentation via code rewriting, as well as some of the concerns a policy writer might have, such as introducing temporary variables, injecting helper functions, etc.[3]

3.2.1 *Client-Side Error Reporting*. Currently, Web application developers have almost no visibility into errors occurring within users' browsers. Modern JavaScript browsers do allow JavaScript code to provide a custom error handler by setting the `onerror` property:

```
window.onerror = function(msg, file, line){...}
```

However, very few Web applications use this functionality to report errors back to developers. AjaxScope makes it easy to correct this oversight by

[3]Readers familiar with the basics of JavaScript and source-level instrumentation may want to skip to Sections 5–7 for examples of more sophisticated rewriting policies.

automatically augmenting the `onerror` handler to log error messages. For example, a policy may automatically augment registered error handlers without requiring any input from the application developer, resulting in the following code:

```
window.onerror = function(msg, file, line){
    ajaxScopeSend('Detected an error: ' + msg +
            ' at ' + file + ':' + line +
            '\nStack: ' + getStack());
    ... // old handler code is preserved
}
```

One of the shortcomings of the `onerror` handler is the lack of access to a call stack trace and other context surrounding the error. In Section 5.2, we describe how to collect call stack information as part of the performance profile of an application. This instrumentation provides critical context when reporting errors.

3.2.2 *Detecting Potential Infinite Loops.* While infinite loops might be considered an obvious bug in many contexts, JavaScript's dynamic scripts and the side effects of DOM manipulation make infinite loops in complex AJAX applications more common than one might think.

Example 1. The following code shows one common pattern leading to infinite loops [Zakas et al. 2006].

```
for (var i=0; i < document.images.length; i++) {
    document.body.appendChild(
        document.createElement("img"));
}
```

The array `document.images` grows as the body of the loop is executing because new images are being generated and added to the body. Consequently, the loop never terminates.

To warn a Web developer of such a problem, we automatically instrument all `for` and `while` loops in JavaScript code to check whether the number of iterations of the loop exceeds a developer-specified threshold. While we cannot programmatically determine that the loop execution will never terminate, we can reduce the rate of false positives by setting the threshold sufficiently high. Below we show the loop above instrumented with infinite loop detection:

```
var loopCount = 0, alreadySent = false;
for (var i = 0; i < document.images.length; i++) {
if (!alreadySent &&
    (++loopCount > LOOP_THRESHOLD)) {
    ajaxScopeSend('Unexpectedly long loop '
    + ' iteration detected');
    alreadySent = true;
}
document.body.appendChild(
    document.createElement('img'));
}
```

When a potential infinite loop is detected, a warning message is logged for the Web application developer. Such a warning could also trigger extra instrumentation to be added to this loop in the future to gather more context about why it might be running longer than expected. This example injects new temporary variables `loopCount` and `alreadySend`; naming conflicts can be avoided using methods proposed in BrowserShield for tamper-proofing [Reis et al. 2006].

3.2.3 *Detecting Inefficient String Concatenation.* Because string objects are immutable in JavaScript, every string manipulation operation produces a new object on the heap. When concatenating a large number of strings together, avoiding the creation of intermediate string objects can provide a significant performance improvement, depending on the implementation of the JavaScript engine. One way to avoid generating intermediate strings is to use the native JavaScript function `Array.join`, as suggested by several JavaScript programming guides [Internet Explorer development team; Crisp 2006]. Our own microbenchmarks, shown in Table I, indicate that using `Array.join` instead of the default string concatenation operator + can produce over 130x performance improvement on different versions of IE.

Example 2. The string concatenation in the following code

```
var small = /* Array of many small strings */;
var large = '';
for (var i = 0; i < small.length; i++) {
    large += small[i];
}
```

executes more quickly on Internet Explorer 6, Internet Explorer 7, and FireFox 1.5 if written as: `var large = small.join('')`.

To help discover opportunities to replace the + operator with `Array.join` in large programs, we instrument JavaScript code to track string concatenations. To do so, we maintain “depth” values, where depth is the number of string concatenations that led to the creation of a particular string instance. The depth of any string not generated through a concatenation is 0. Our instrumentation rewrites every concatenation expression of the form `a = b + c`, where `a` is a variable reference, and `b` and `c` are expressions. The rewritten form is:

```
var tmp1,tmp2;
...
(tmp1 = b, tmp2 = c, tmp3 = a,
    a = tmp1 + tmp2,
    adjustDepth(tmp1, tmp2, tmp3), a)
```

where the comma operator in JavaScript is used to connect statements. We use a helper function `adjustDepth` to dynamically check that the types of `b` and c are strings, to compute the maximum depth of b and c increased by 1 and associate it with a.[4] Depth maintenance is accomplished by having a global hash map of `string` $\rightarrow$ `depth` values.[5] Whenever the depth first exceeds a user-defined threshold, a warning message is logged. This instrumentation goes beyond pattern-matching in simple loops, finding opportunities to optimize even interprocedural string concatenations.

3.3 Structure of an Instrumentation Policy

In designing AjaxScope, we have found it helpful to give some structure to the policies we were developing. In particular, the description below motivates the idea of pipelining policies together. To describe the structure of an instrumentation policy in AjaxScope, we first present some key definitions.

—An *instrumentation point* is any instance of a language element in JavaScript code, such as a function declaration, statement, variable reference, or the program as a whole. Instrumentation points are represented as abstract syntax tree (AST) nodes of the JavaScript program's parse tree.

—*Policy nodes* are the basic unit of organization for analyzing and instrumenting JavaScript code. The primary purpose of a policy node is to rewrite the instrumentation point to report observations of its runtime state and/or apply a static or runtime analysis. We discuss policy nodes in more detail in Section 3.4.

—Policy nodes are pipelined together to form a complete *instrumentation policy*. This pipeline represents a dataflow of instrumentation points from one policy node to the next. The first instrumentation point entering the pipeline is always the root AST node of a JavaScript program.

The JavaScript rewriting examples presented in Section 3.2 are all instrumentation policies implementable with a simple two-stage pipeline, as shown in Figure 2.

Two components within the AjaxScope proxy provide key support functionality for instrumentation policies. The *parser* is responsible for identifying and extracting JavaScript code from the HTTP traffic passing through the proxy. Once identified, the JavaScript code is parsed into an abstract syntax tree (AST) representation and passed through each of the instrumentation policies. The *log collector* receives and logs messages reported by instrumentation code embedded within a Web application and distributes them to the responsible instrumentation policy for analysis and reporting.

3.4 Policy Nodes

To support their analysis of code behavior, policy nodes may maintain global state or state associated with an instrumentation point. One of the simplest

[4]For this rewriting, function `adjustDepth` is injected by AjaxScope into the header of every translated page.
[5]Since strings are passed by value in JavaScript, this approach can occasionally result in false positives, although we have not seen that in practice.

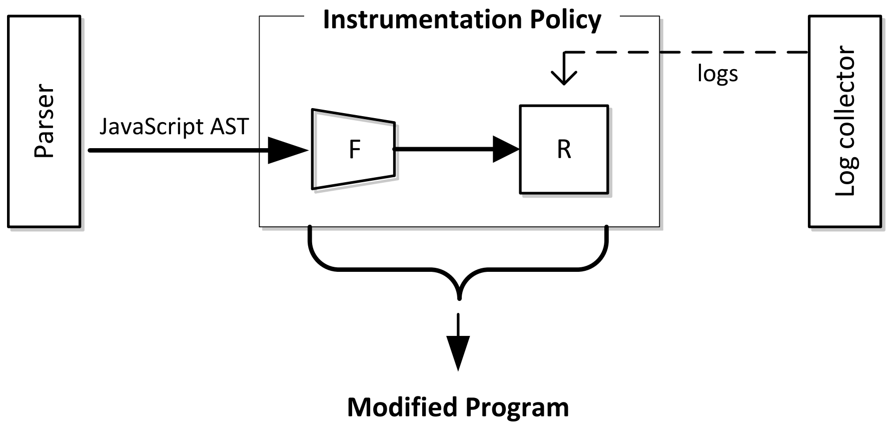


Fig. 2. Structure of an instrumentation policy. The first `F` policy node in these policies is a simple static analysis or filter to identify relevant instrumentation points. The second `R` policy node is a rewriting node to inject instrumentation code into the program.

kinds of policy nodes are stateless and stateful *filters*. Stateless filter nodes provide the simple functionality of a search through an abstract syntax tree. Given one or more AST nodes as input, a filter node will search through the tree rooted at each AST node, looking for any instrumentation point that matches some constant filter constraints. The results of this search are immediately output to the next stage of the policy pipeline.

A stateful filter searches through an AST looking for instrumentation points that not only match some constant filter constraint, but which are also explicitly flagged in the filter's state. This stateful filter is a useful primitive for enabling human control over the operation of a policy. It is also useful for creating a feedback loop within a policy pipeline, allowing policy nodes later in the pipeline to use potentially more detailed information and analysis to turn on or off the instrumentation of a code location earlier in the pipeline.

Some policy nodes may modify their injected instrumentation code or their own dynamic behavior based on this state. We describe one simple and useful form of such adaptation in Section 5. Policy nodes have the ability to inject either varied or uniform instrumentation code across users. We describe how we use this feature to enable distributed tests and A/B tests in Sections 6 and 7.

4. IMPLEMENTATION

We have implemented an AjaxScope proxy prototype as described in this article. Sitting between the client browser and the servers of a Web application, the AjaxScope proxy analyzes HTTP requests and responses and rewrites the JavaScript content within, according to instantiated instrumentation policies.

Our AjaxScope prototype implementation consists of 4 major code modules:

—The core platform, responsible for coordinating among instrumentation policies and logging, as described in Section 3.1.

Table III. Lines of Code for Code Modules of AjaxScope Proxy Implementation

Module	**Lines of code**
Policy and logging platform	2,900
JavaScript parser	25,000
HTTP and HTML handling.	3,900
Various policies	4,500
Total	36,300

—The *JavaScript parser* module, based on the ECMA language specification [ECMA 1999]. The performance of our parser primarily determines the performance of our instrumentation, and is detailed further in Section 4.1.

—The *HTTP and HTML handling module* is responsible for the low-level proxy functionality and HTML parsing. Our implementation relies on the kernel-level HTTP subsystem within the Windows operating system for most of its HTTP handling functionality.

—The various instrumentation policies, including the example policies described in Section 3.2 the adaptive performance instrumentation policies described in Section 5, and the distributed memory leak checking described in Section 6. The entire AjaxScope prototype is implemented in C#.

Table III details the size of the individual modules. Our prototype includes some straightforward performance optimizations, such as caching the parsed representation of JavaScript snippets to avoid reparsing the same script on every reload of a page. However, our implementation has not been optimized to scale to medium or large usage. Most significantly, for a large Web site, we would recommend that instrumentation and rewriting of JavaScript code *not* be done on a per-request basis. Instead, we would recommend that application code be rewritten periodically, such as every 20 seconds. Such a change would avoid adding parsing and instrumentation overhead to the critical path of serving Http requests to a large audience.

Although we have not evaluated it in this article, we have also implemented a server-side version of AjaxScope that integrates with Microsoft IIS (Web server). In the server-side implementation, the Web application's hosting Web server is responsible for dynamic instrumentation and log collection. This implementation is available for download as Microsoft Visual Studio AJAX Profiling Extensions.[6]

4.1 Microbenchmarks

Before we characterize overhead numbers for large Web applications, we first present some measurements of aspects of AjaxScope that affect almost every instrumentation.

Logging Overhead. By default, instrumented Web applications queue their observations of application behavior in memory. Our instrumentation schedules a timer to fire periodically, collating queued messages and reporting them back to the AjaxScope proxy via an HTTP `POST` request.

[6]http://code.msdn.microsoft.com/AjaxView

Table IV. Overhead of Message Logging Across Browsers. The Test Code Measured in this Experiment and Calls a Function 10k Times, Causing 20k Logging Messages to be Recorded. All Times are Reported in Msec

Browser	**w/out Instrumentation**		**w/Instrumentation**		**Per-message**
	mean	**std.dev.**	**mean**	**std.dev.**	**overhead**
IE 7.0	80	30	407	40	0.016
FireFox 1.5	33	14	275	40	0.012

To assess the critical path latency of logging a message within an instrumented program, we wrote a simple test case program that calls an empty JavaScript function in a loop 10,000 times. With function-level instrumentation described in Section 5.2, there are two messages that are logged for each call to the empty function. As a baseline, we first measure total execution time of this loop without instrumentation and then measure with instrumentation. We calculate the time to log a single message by dividing the difference by the 2×10^4 number of messages logged. We ran this experiment 8 times to account for performance variations related to process scheduling, caching, etc. As shown in Table IV, our measurements show that the overhead of logging a single message is approximately 0.01–0.02 ms.

Parsing Latency. We find that the parsing time for our unoptimized AjaxScope JavaScript parser is within an acceptable range for the major Web 2.0 sites we tested. In our measurements, parsing time grows approximately linearly with the size of the JavaScript program. It takes AjaxScope about 600 msec to parse a 10,000-line JavaScript file. Dedicated server-side deployments of AjaxScope can improve performance with cached AST representations of JavaScript pages.

Note that our parser is not a streaming parser: it waits until the entire program or page has been arrived to commence the parsing process. Converting to a streaming parser might be a more efficient way to go. However, note that in real-life deployments, the process of code parsing and rewriting is done offline, so the latency of the parsing process does not affect the end-user. In practice, precomputed instrumented versions may be needed for different browser/language combinations, so there may be need for an initial stub that dispatches the user to the correct version. However, we already see this kind of specialization with real-life sites: the same site may have different versions customized for desktop and mobile browsers.

4.2 Experimental Setup

For our experiments (presented in Sections 5–7) we used two machines connected via a LAN hub to each other and the Internet. We set up one machine as a proxy running our AjaxScope prototype. We set up a second machine as a client, running various browsers configured to use AjaxScope as a proxy. The proxy machine was an Intel dual core Pentium 4, clock rate 2.8GHz with 1GB of RAM, running Windows Server 2003/SP1. The client machine was an Intel Xeon dual core, clock rate 3.4GHz with 2.5GB of RAM, running Windows XP/SP2.

4.3 Benchmark Selection

We manually selected 12 popular Web applications and sites from several categories, including portals, news sites, games, etc. Summary information about our benchmarks is shown in Table V. This information was obtained by visiting the page in question using either Internet Explorer 7.0 or Mozilla FireFox 1.5 with AjaxScope's instrumentation. We observed overall execution time with minimal instrumentation enabled to avoid reporting instrumentation overhead. Separately, we enabled fine-grained instrumentation to collect information on functions executed during page initialization.

Most of our applications contain a large client-side component, shown in the code size statistics. There are often small variations in the amount of code downloaded for different browsers. More surprising is the fact that even during the page initialization alone, a large amount of JavaScript code is getting executed, as shown by the runtime statistics. As this initial JavaScript execution is a significant component of page loading time as perceived by the user, it presents an important optimization target and a useful test case. For the experiments in Sections 5–6, we use these page initializations as our test workloads. In Section 7, we use manual searching and browsing of Live Maps as our test application and workload.

In addition to the 12 Web applications already described, we also benchmark 78 other Web sites. These sites are based on a sample of 100 URLs from the top 1 million URLs clicked on after being returned as MSN Search results in Spring 2005. The sample of URLs is weighted by click-through count and thus includes both a selection of popular Web sites as well as unpopular or tail Web sites. From these 100 URLs, we removed those that either 1) had no JavaScript; 2) had prurient content; 3) were already included in the 12 sites described above; or 4) were no longer available.

4.4 Overview of Experiments

In subsequent sections, we present more sophisticated instrumentation policies and use them as examples to showcase and evaluate different aspects of AjaxScope. The next section describes issues of policy adaptation, using drill-down performance profiling as an example. Section 6 describes distributed policies, using a costly memory leak detection policy as an example. Finally, Section 7 discusses the function result caching policy, an optimization policy that uses A/B testing to dynamically evaluate the benefits of cache placement decisions.

5. ADAPTIVE INSTRUMENTATION

This section describes how we build adaptive instrumentation policies and how such adaptive instrumentation can be used to reduce the performance and network overhead of function-level performance profiling, via *drill-down* performance profiling.

Table V. Benchmark Application Statistics for Internet Explorer 7.0 (IE) and FireFox 1.5 (FF)

	STATIC						RUNTIME (PAGE INITIALIZATION)							
Web application or site	**JavaScript code size**						**Number of functions**						**Execution time (ms)**	
	LoC		**KB**		**Files**		**Declared**		**Executed**		**Unique Ex.**			
Mapping services														
`maps.google.com`	33511	33511	295	295	7	7	1935	1935	17587	17762	618	616	530	610
`maps.live.com`	63787	65874	924	946	6	7	2855	2974	4914	4930	577	594	190	150
Portals														
`msn.com`	11,499	11,603	124	127	10	11	592	592	1,557	1,557	189	189	301	541
`yahoo.com`	18,609	18,472	278	277	5	5	1,097	1,095	423	414	107	103	669	110
`google.com/ig`	17,658	17,705	135	167	3	3	960	960	213	213	59	59	188	244
`protopages.com`	34,918	35,050	599	599	2	2	1,862	1,862	0	0	0	0	13,782	1,291
News sites														
`cnn.com`	6,299	6,473	126	137	24	25	197	200	120	139	56	63	234	146
`abcnews.com`	7,926	8,004	121	122	20	21	225	228	810	810	86	86	422	131
`bbcnews.co.uk`	3,356	3,355	57	57	10	10	142	142	268	268	23	23	67	26
`businessweek.com`	7,449	5,816	135	119	18	13	258	194	7,711	7,711	137	137	469	448
Online games														
`chi.lexigame.com`	9,611	9,654	100	100	2	2	333	333	769	769	55	55	208	203
`minesweeper.labs.morfik.com`	33,045	34,353	253	265	2	2	1,210	1,210	290	290	122	122	505	650

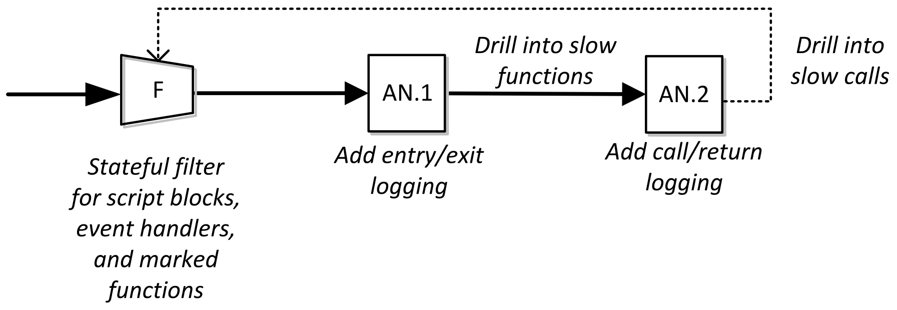

F
AN.1
AN.2
Drill into slow functions
Drill into slow calls
Stateful filter for script blocks, event handlers, and marked functions
Add entry/exit logging
Add call/return logging

Fig. 3. Policy for drill-down performance profiling.

5.1 Adaptation Nodes

Adaptation nodes are specialized policy nodes which take advantage of the serial processing by instrumentation policy nodes to enable a policy to have different effects over time. The key mechanism is simple: for each instrumentation point that passes through the pipeline, an adaptation node makes a decision to either instrument the node itself or to to pass the instrumentation point to the next policy node for instrumentation. Initially, the adaptation node applies its own instrumentation and then halts the processing of the particular instrumentation point, sending the instrumentation point in its current state to the end-user. In later rounds of rewriting, for example, when other users request the JavaScript code, the adaptation node will revisit this decision. For each instrumentation point, the adaptation node will execute a specified test and, when the test succeeds, allow the instrumentation point to advance and be instrumented by the next adaptation node in the policy.

5.2 Naïve Performance Profiling

One naïve method for performance profiling JavaScript code is to simply add timestamp logging to the entry and exit points of every JavaScript function defined in a program. Calls to native functions implemented by the JavaScript engine or browser (such as DOM manipulation functions, and built-in mathematical functions) can be profiled by wrapping timestamp logging before and after every function call expression. However, because this approach instruments every function in a program, it has a very high overhead, both in added CPU time as well as network bandwidth for reporting observations.

5.3 Drill-Down Performance Profiling

Using AjaxScope, we have built an adaptive, drill-down performance profiling policy, shown in Figure 3, that adds and removes instrumentation to balance the need for measuring the performance of slow portions of the code with the desire to avoid placing extra overhead on already-fast functions.

Initially, our policy inserts timestamp logging 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). Once this coarse-grained instrumentation gathers enough information to identify slow script blocks and event handlers, the policy adds additional instrumentation to discover the performance of the functions that are being called by each slow script block and event handler. As clients download and execute fresh copies of the application, they will report more detail on the performance of the slow portions of code.

After this second round of instrumentation has gathered enough information, our policy drills down once again, continually searching for slower functions further down the call stack. To determine when to drill down into a function, we use a simple nonparametric test to ensure that we have collected enough samples to be statistically confident that our observed performance is higher than a given performance threshold. In our experiments, we drill down into any function believed to be slower than 5 msec. Eventually, the drill-down process stabilizes, having instrumented all the slow functions, without ever adding any instrumentation to fast functions.

5.4 Evaluation

The goal of our adaptation nodes is to reduce the CPU and network overhead placed on end-user's browsers by brute-force instrumentation policies while still capturing details of bottleneck code. To measure how well our adaptive drill-down performance profiling improves upon the naïve full performance profiling, we tested both policies against our 90 benchmark applications and sites. We first ran our workload against each Web application 10 times, drilling down into any function slower than 5 msec. After these 10 executions of our workload, we captured the now stable list of instrumented function declarations and function calls and measured the resulting performance overhead. Our full performance profiler simply instrumented every function declaration and function call. We also used a minimal instrumentation policy, instrumenting only high-level script blocks and event handlers, to collect the base performance of each application.

Figure 4 shows how using adaptive drill-down significantly reduces the number of instrumentation points that have to be monitored in order to capture bottleneck performance information. While full-performance profiling instruments a median of 89 instrumentation points per application (mean=129), our drill-down profiler instruments a median of only 3 points per application (mean=3.7).

This reduction in instrumentation points—from focusing only on the instrumentation points that actually reveal information about slow performance—also improves the execution and network overhead of instrumentation and log reporting. Figures 5 and 6 compare the execution time overhead and logging message overhead of full performance profiling and drill-down performance profiling on FireFox 1.5 (graphs of overhead on Internet Explorer 7 are almost identical in shape). Table VI shows the median and 90th percentile performance of our benchmark Web sites under no instrumentation, full profiling and

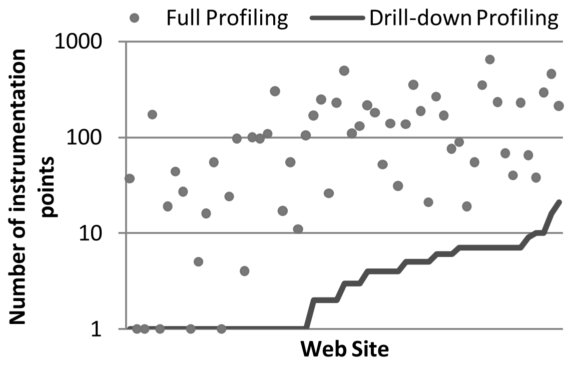


Fig. 4. Number of functions instrumented per Web site with full profiling vs. drill-down profiling.

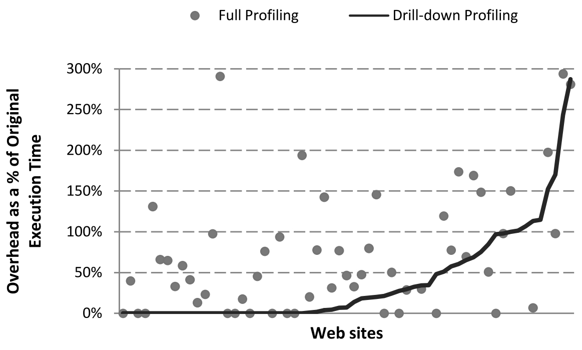


Fig. 5. Execution time overhead of drill-down performance profiling compared to full performance profiling.

drill-down profiling. At the 90th percentile, we find that adaptive instrumentation provides an improvement of over 150 msec. Amazon.com has reported the importance it places on performance even at the 99.9th percentile [DeCandia et al. 2007]; Google reports that such performance differences can have a noticeable effect on user behavior [Brutlag 2009].

As seen in Figure 5, seven of our 90 sites appear to show better performance under full profiling than drill-down profiling. After investigation, we found that 5 of these sites have little JavaScript executing, and the measured difference in overhead is within the approximate precision of the JavaScript timestamp (around 10–20 msec). Due to an instrumentation bug, 1 site failed when full instrumentation was enabled, resulting in a measurement of a very low overhead. The 7th site appears to be a legitimate example where full profiling is faster than drill-down profiling. This could be due to subtle differences in the injected instrumentation or, though we attempted to minimize such effects, it may be due to other processes running in the background during our drill-down profiling experiment.

While overall the reduction in CPU overhead was modest, the mean network overhead from log messages improved substantially, dropping from 300KB to

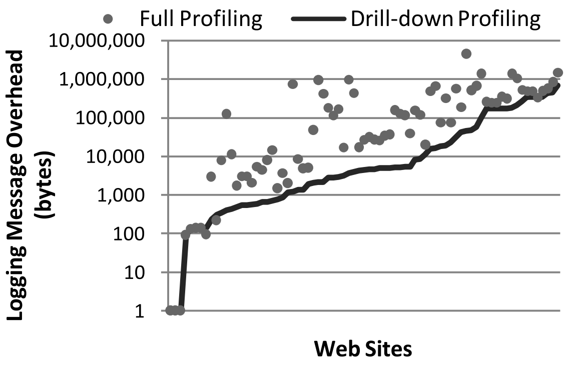


Fig. 6. Logging overhead of drill-down performance profiling compared to full performance profiling.

Table VI. The Median and 90th Percentile Performance, in msec, of Our 90 Benchmark Applications and Sites Without Instrumentation, with Full Profiling and With Drill-down Profiling

	Baseline	**Full profiling**	**Drill down profiling**
Median	28	37	31
90th	130	301	148

64KB, and the median overhead dropping from 92KB to 4KB. This improvement is particularly important for end-users sitting behind slower asymmetric network links.

6. DISTRIBUTED INSTRUMENTATION

This section describes how we build distributed instrumentation policies in AjaxScope, and then applies this technique to reduce the per-client overhead of an otherwise prohibitively expensive memory leak checker.

6.1 Distributed Tests

The second specialized policy node we provide as part of the AjaxScope platform is the *distributed test*. A distributed test is a runtime analysis testing some property of the program's execution, that spreads out the overhead of instrumentation code across many users' executions of a Web application. Note that all distributed tests are also adaptation nodes, since distributed tests cannot evaluate until gathering observations of runtime behavior.

At any given point in time, the value of the distributed test can be in one of three states with respect to a specific instrumentation point: (1) *pass*, the instrumentation point has passed the test, in which case it will be sent to the next policy node in the pipeline; (2) *fail*, the instrumentation point has failed the test, in which case, it will not be sent to the next policy node and the distributed test will cease instrumenting it; and (3) *more testing*, the information gathered so far is insufficient and the distributed test needs to gather further observations.

Our distributed test abstraction requires that a policy writer provide the specific rewriting rule that measures some runtime behavior of an application, and the parameters to a simple test function. However, once this is provided, the distributed test provides for the randomized distribution of instrumentation across the potential instrumentation points and users, and the evaluation of the test for each instrumentation point.

Our AjaxScope prototype provides distributed tests on pseudo-Boolean as well as numerical measures. In the pseudo-Boolean case, we allow the measure of runtime behavior to return one of 5 values: `TotalFailure`, `Failure`, `Neutral`, `Success`, `TotalSuccess`. If a measure at an instrumentation point ever reports a `TotalFailure` or `TotalSuccess`, the distributed test value for that point is immediately set to fail or pass, respectively. If neither a `TotalFailure` nor `TotalSuccess` have been reported, then the parameterized test function is applied to the number of failure, neutral, and success observations. In the case of numerical measures, the distributed test directly applies the parameterized test function to the collection of metrics.

A more advanced implementation of distributed tests would dynamically adjust the rate at which different instrumentation points were rewritten, for example, to more frequently instrument the rare code paths and less frequently instrument the common code path [Hauswirth and Chilimbi 2004]. We leave such an implementation to our future work.

6.2 Memory Leaks in AJAX Applications

Memory leaks in JavaScript have been a serious problem in Web applications for years. With the advent of AJAX, which allows the same page to be updated numerous times, often remaining in the user's browser for a period of hours or even days, the problem has become more severe. Despite being a garbage-collected language, JavaScript still suffers from memory leaks. One common source of such leaks is the failure to nullify references to unused objects, making it impossible for the garbage collector to reclaim them [Shaham et al. 2002]. Other memory leaks are caused by browser implementation bugs [Schlueter 2006; Baron 2001].

Here, we focus on a particularly common source of leaks: cyclical data structures that involve DOM objects. JavaScript interpreters typically implement the mark-and-sweep method of garbage collection, so cyclical data structures *within the JavaScript heap* do not present a problem. However, when a cycle involves a DOM element, the JavaScript collector can no longer reclaim the memory, because the link from the DOM element to JavaScript "pins" the JavaScript objects in the cycle. Because of a reference from JavaScript, the DOM element itself cannot be reclaimed by the browser. This problem is considered a bug in Web browsers and has been fixed or mitigated in the latest releases. However, it remains a significant issue because of the large deployed base of older browsers.

Because these leaks can be avoided through careful JavaScript programming, we believe it is a good target for highlighting the usefulness of dynamic monitoring. Note that these leaks might be tricky to find using static analysis

```
<html>
<head>
<script type="text/javascript">
    var globalObj = new Object;

    function SetupLeak(){
        globalObj.foo = document.getElementById("leaked");
        document.getElementById("leaked").
                                someProperty = globalObj;
    }
</script>
</head>
<body onload="SetupLeak()">
<div id="leaked"></div>
</body>
</html>
```

Fig. 7. An object cycle involving JavaScript and DOM objects.

because DOM data structures are highly recursive, which presents a problem for static analysis precision: most scalable static analyses would represent a complex data structure like a fragment of the DOM with just one static node, leading to false warnings. Despite some degree of success with static analysis of JavaScript in other areas [Guarnieri and Livshits 2009], we believe runtime analysis to be a better match for this particular problem.

Example 3. An example of such a memory leak is shown in Figure 7. DOM element whose DOM `id` is `leaked` has a pointer to the global JavaScript object `globalObj` through property `someProperty`. Conversely, `globalObj` has a pointer to `leaked` through property `foo`. The link from `leaked` makes it impossible to reclaim `global`; at the same time the DIV element cannot be reclaimed since `global` points to it.

Explicit cycles such as the one in Figure 7 are not the most common source of leaks in real applications, though. JavaScript closures inadvertently create these leaking cycles as well.

Example 4. Figure 8 gives a typical example of closure misuse, leading to the creation of cyclical heap structures. The DOM element referred to by `obj` points to the closure through its `onclick` property. At the same time, the closure includes implicit references to variables in the local scope so that references to them within the closure function body can be resolved at runtime. In this case, the event handler function will create an implicit link to `obj`, leading to a cycle. If this cycle is not explicitly broken before the Web application is unloaded, this cycle will lead to a memory leak.

6.3 Instrumentation

To detect circular references between JavaScript and DOM objects, we use a straight-forward, brute-force runtime analysis of the memory heap. First, we

```
<html>
<head>
<script type="text/javascript">
    window.onload = function(){
    var obj = document.getElementById("element");
    obj.onclick = function(evt){ ... };
};
</script>
</head>
<body><div id="element"></div></body>
</html>
```

Fig. 8. A memory leak caused by erroneous use of closures.

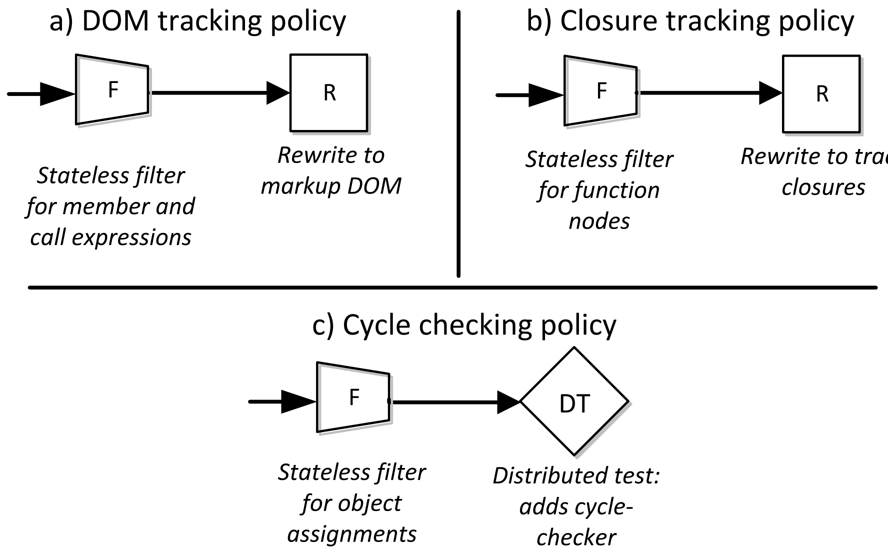

a) DOM tracking policy
F
R
Stateless filter for member and call expressions
Rewrite to markup DOM
b) Closure tracking policy
F
R
Stateless filter for function nodes
Rewrite to track closures
c) Cycle checking policy
F
DT
Stateless filter for object assignments
Distributed test: adds cycle-checker

Fig. 9. Three instrumentation policy pipelines work together to catch circular references between DOM and JavaScript objects that are potential memory leaks.

use one instrumentation policy to dynamically mark all DOM objects. A second instrumentation policy explicitly tracks closures, so that we can traverse the closure to identify any circular references caused inadvertently by closure context. Finally, a third instrumentation policy instruments all object assignments to check for assignments that complete a circular reference. This last policy is the instrumentation that places the heaviest burden on an end-user's perceived performance. Thus, we implement it as a distributed test to spread the instrumentation load across users. Figure 9 illustrates these three instrumentation policies.

6.3.1 *Marking DOM objects.* We mark DOM objects returned from methods `getElementById`, `createElementById`, and other similar functions as well as objects accessed through fields such as `parentNode`, `childNodes`, etc. The

```
(tmp1 = div,
tmp2 = document.getElementById("leaked"),
tmp2.isDOM=true,
tmp2.sub = tmp1,
checkForCycles(tmp1, tmp2,
  'Checking document.getElementById("leaked").sub=div');
```

Fig. 10. Code rewriting for DOM object marking.

marking is accomplished by setting the `isDOM` field of the appropriate object. For example, assignment

```
var obj = document.getElementById("leaked");
```

in the original code will be rewritten as

```
var tmp; var obj=(tmp=document.getElementById("leaked"),
         tmp.isDOM = true, tmp);
```

As an alternative to explicitly marking DOM objects, we could also have speculatively infer the type of an object based on whether it contained the members of a DOM object.

6.3.2 *Marking Closures.* Since closures create implicit links to the locals in the current scope, we perform rewriting to make these links explicit, so that our detection approach can find cycles. For instance, the closure creation in Figure 8 will be augmented in the following manner:

```
obj.onclick = (tmp = function(evt){ ... },
  tmp.locals = new Object, tmp.locals.l1 = obj, tmp);
```

This code snippet creates an explicit link from the closure assigned to `obj.onclick` to variable `obj` declared in its scope. The assignment to `obj.onclick` will be subsequently rewritten as any other store to include a call to helper function `checkForCycles`. This allows our heap traversal algorithm to detect the cycle

$$\texttt{function(evt)\{...\}} \rightarrow \texttt{function(evt)\{...\}.locals} \rightarrow \\ \texttt{obj} \rightarrow \texttt{obj.onclick}$$

6.3.3 *Checking Field Stores for Cycles.* We check all field stores of JavaScript objects to determine if they complete a heap object cycle that involves DOM elements. For example, field store

```
document.getElementById("leaked").sub = div;
```

will be rewritten as shown in Figure 10.

Finally, an injected helper function, `checkForCycles`, performs a depth-first heap traversal to see if (1) `tmp2` can be reached by following field accesses from `tmp1` and (2) if such a cycle includes a DOM object, as determined by checking the `isDOM` property, which is set as described above.

```
var tmp;
var obj=(tmp=document.getElementById("leaked"),
         tmp.isDOM = true, tmp);
var pipelineContainers = document.
     getElementById("cnnPipelineModule").getElementsByTagName("div");
...
for (var i=0; i<pipelineContainers.length; i++){
    var pipelineContainer = pipelineContainers[i];
    if(pipelineContainer.id.substr(0,9) == "plineCntr") {
        pipelineContainer.onmouseover = function () {
             CNN_changeBackground(this,1); return false;}
        pipelineContainer.onmouseout = function () {
             CNN_changeBackground(this,0); return false;}
    }
... }
```

Fig. 11. A circular reference in `cnn.com`, file `mainVideoMod.js` (edited for readability). Unless this cycle is explicitly broken before page unload, it will leak memory.

6.4 Evaluation

As with our adaptation nodes, the goal of our distributed tests is to reduce the overhead seen by any single user, while maintaining aggregate visibility into the behavior of the Web application under real workloads. To distribute our memory checking instrumentation, we implement our field store cycle check as a distributed test, randomly deciding with some probability whether to add this instrumentation to any given instrumentation point. We continue to uniformly apply the DOM tracking and closure tracking policies. In our experiments, the overhead added by these two policies was too small to measure.

Applying our memory leak checker, we found circular references indicating a potential memory leak in the initialization code of 4 of the 12 JavaScript-heavy applications in our benchmarks, including `google.com/ig`, `yahoo.com`, `chi.lexigame.com`, and `cnn.com`.

Example 5. As a specific example of a potential memory leak, Figure 11 shows code from the video player located on the `cnn.com` main page, where there is a typical memory leak caused by closures. Here, event handlers `onmouseover` and `onmouseoout` close over the local variable `pipelineContainer` referring to a `div` element within the page. This creates an obvious loop between the `div` and the closure containing handler code, leading to a leak.

Figure 12 shows a histogram of the performance overhead of an individual cycle check, graphed on a log-scale y-axis. We see that almost all cycle checks have a minimal performance impact, with a measured overhead of 0msec. A few cycle checks do last longer, in some cases up to 75 msec. We could further limit this overhead of individual cycle checks by implementing a random walk of the memory heap instead of a breadth-first search. We leave this to future work.

To determine whether distributing our memory leak checks truly reduced the execution overhead experienced by users, we loaded `cnn.com` in Internet

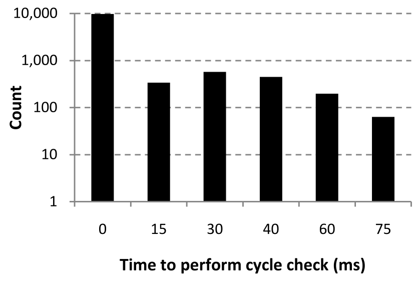


Fig. 12. The histogram of circular reference check times. The y-axis is log-scale. The vast majority of checks for cycles take under 1 msec.

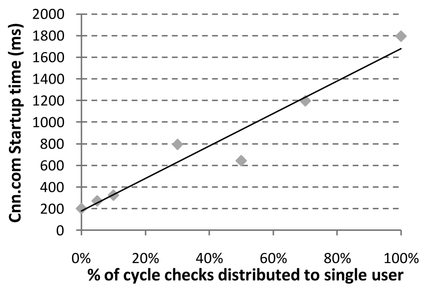


Fig. 13. The average startup time for `cnn.com` increases linearly with the probability of injecting a cycle check.

Explorer 7 with varying probabilities of instrumentation injection, measured the time to execute the page's initialization code, and repeated this experiment several times each for different probability settings. Figure 13 shows the result and we can see that, as expected, the average per-user overhead is reduced linearly as we reduce the probability of injecting cycle checks into any single user's version of the code. At a probability of 100%, we are adding 1,600 cycle checks to the Web application, resulting in an average startup time of 1.8 sec. At 0% instrumentation probability, we reduce the startup to its baseline of 230 msec. This demonstrates that simple distribution of instrumentation across users can turn heavyweight runtime analyses into practical policies with a controlled impact on user-perceived performance.

7. A/B TESTING

On Web sites, A/B testing is commonly used to evaluate the effect of changes to banner ads, newsletters, or page layouts on user behavior. In our developer-oriented scenarios, we use A/B tests to evaluate the performance impact of a specific rewriting, such as the code optimization described in Section 3.2. The A/B test policy node serves the original code point to $X\%$ of the Web application's users and serves the rewritten version of the code point to the other $(100 - X)\%$ of users. In both cases, the A/B test adds instrumentation to measure the performance of the code point. The resulting measurements allow

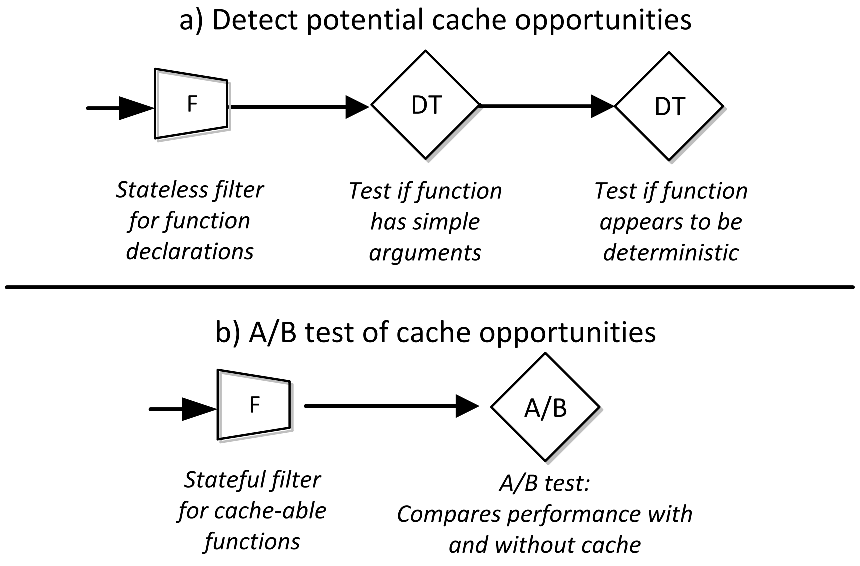

a) Detect potential cache opportunities
F
DT
DT
Stateless filter for function declarations
Test if function has simple arguments
Test if function appears to be deterministic
b) A/B test of cache opportunities
F
A/B
Stateful filter for cache-able functions
A/B test: Compares performance with and without cache

Fig. 14. Two policies work together for detection and performance testing of cache opportunities. After policy (a) finds a potential cache opportunity, a human developer must check its semantic correctness before policy (b) can test it for performance improvements.

us to evaluate the average performance improvement, as well as the average improvements for a subpopulation of users, such as all FireFox users. A more advanced implementation of A/B tests could potentially monitor the rates of exceptions occurring within the block of code, to notice potential reliability issues.

7.1 Function Return Value Caching

With live monitoring, we can use a multistage 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 check the first criteria, and rely on a human developer to understand the semantics of the function to determine the second. Finally, we use a second stage of instrumentation to check whether the benefits of caching outweigh the cost. Both stages of instrumentation are illustrated in Figure 14.

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.

Table VII. Results of Search for Potential Cacheable Functions in Live Maps

Function	Determ.	Hit rate	Performance Original (ms)	Performance Cached (ms)	Improvement ms	Improvement %
`OutputEncode_`						
`EncodeURL`	✓	77%	0.85	0.67	0.18	21%
`DegToRad`	✓	85%	0.11	0.00	0.11	100%
`GetWindowHeight`	x	90%	2.20	0.00	2.20	100%
`GetTaskArea`						
`BoundingBoxOffset`	x	98%	1.70	0.00	1.70	100%
`GetMapMode`	x	96%	0.88	0.00	0.88	100%

After gathering many observations over a sufficient variety and number of user workloads, we provide a list of potentially cacheable 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 nondeterminism. The advantage of this first stage of monitoring is that reviewing a short list of possibly valid cacheable 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 is high, but not otherwise.

7.2 Evaluation

In this section, we described two instrumentation policies: the first searches for potential caching opportunities, while the second tests their performance improvement using automatic comparison testing of the original and optimized versions. The goal of both policies is to reduce the effort developers must make to apply simple optimizations to their code, and to show how dynamic A/B testing can be used to evaluate the efficacy of such optimizations under real-life user workloads.

Table VII shows the end results of applying these policies to `maps.live.com`. Our instrumentation started with 1,927 total functions, and automatically reduced this to 29 functions that appeared to be deterministic. To exercise the application, we manually applied a workload of common map-related activities, such as searching, scrolling, and zooming. Within a few minutes, our A/B test identified 2 caching opportunities that were both semantically deterministic and improved each function's performance by 20%–100%. In addition, we identified 3 relatively expensive functions, including a `GetWindowHeight` function, that empirically appeared to be deterministic in our workloads but semantically are likely to not be deterministic. Seeing these results, our recommendation would be to modify the implementation of these functions to support caching, while maintaining correctness by explicitly invalidating the cache when an event, such as a window size change, occurs. We expect that these kinds of automated analysis and optimization would be even more useful for newly written or beta versions of Web applications, in contrast to a mature, previously optimized application such as Live Maps.

8. MEDIC: USING AJAXSCOPE FOR DEBUGGING

The AjaxScope proxy provides a programmatic framework for instrumenting and collecting data on the behavior of Web applications running inside end-users' browsers. While AjaxScope provides primitives for distributed and adaptive instrumentation, instrumentation policies are responsible for managing any dependencies between instrumentation points as well as managing the runtime execution of instrumentation points. For example, in adaptive performance profiling (Section 5), whenever the beginning of a function is instrumented we must also instrument the end of that function. Such cases of dependent instrumentation can quickly become more complex, however, and may sometimes be spread across different functions or even different JavaScript files. Furthermore these dependencies may encompass both static instrumentation dependencies and runtime dependencies. To provide abstractions to ease developers' monitoring of application behavior and investigation of problems, we have extended AjaxScope with the MEDic system (Monitoring, Evolving, and Debugging Web applications). MEDic consists of two components.

(1) The *Probabilistic Instrumentation Library* supports sampled instrumentation conditioned on both runtime and static dependencies. The library provides out-of-the-box support for call tracing, reporting application state, local variables and function return values, and performance instrumentation.
(2) *StickShift* provides manual, online control of the basic instrumentation of a Web application, as well as a basic visualization of collected data.

The remainder of the section presents a brief introduction to the probabilistic instrumentation library and StickShift.

8.1 Probabilistic Instrumentation Library

In AjaxScope, the distribution of instrumentation code to Web clients is determined as the Web application code is being served. But for some purposes, whether or not instrumentation should be executed is best determined during the execution of the Web application itself. The probabilistic instrumentation library abstracts this ability to determine at execution time whether an instrumentation point will run. Log messages generated by injected instrumentation can be either reported to the server immediately or queued within the Web application for later reporting. Also, queued logs can be conditionally reported based on the outcome of some instrumentation.

```
1  function onMouseDown(event) {
2    if( isDraggable(event.target)) {
3      beginDrag();  // instrumentation point
4    }
5  }
6  function onMouseUp(event) {
7    if( isDraggable(event.target)) {
8      endDrag();    // instrumentation point
9    }
10 }
```

Fig. 15. Sample code showing two dependent instrumentation points (lines 3 and 8). MEDic will instrument both of these points in a single execution, or will instrument neither.

A policy using this library to instrument the code in Figure 15 could, for example, add instrumentation to the calls to `beginDrag()` and `endDrag()`, and control not only how frequently the instrumentation is added to the code, but also how frequently the instrumentation is executed at runtime. Furthermore, the probabilistic instrumentation library allows the policy to specify a dependency between these two instrumentation points stating that the instrumentation of `endDrag()` should not run unless the instrumentation of `beginDrag()` has already executed. Finally, the policy can specify that the instrumentation at `endDrag()` should run dependent on the result of the instrumentation at `beginDrag()`.

In the context of monitoring and debugging Web applications, dependencies between instrumentation points can be used to selectively drill down into the execution of the application. If an error or performance problem occurs only during or after certain user actions or when the application is in a certain state, then later instrumentation can be conditioned to execute only when the necessary conditions are met. This reduces the local performance impact of instrumentation code that need not run.

In contrast to conditional instrumentation, conditional reporting of collected logs is useful in debugging scenarios where either the scenario or trigger of a problem is unknown. For example, if a Web application is known to occasionally fail at a particular point in the code, detailed instrumentation code can be taken leading up to this known failure point. Once the failure point has been reached, the logs generated by the instrumentation can either be thrown away or reported to the server, based on whether a failure actually occurred.

8.2 Stick Shift: Visualization Tool

The Stick Shift tool provides manual, online control over the instrumentation of an application, and a basic visualization of the results. The purpose of Stick Shift is to give developers a simple way to monitor, explore and debug the current behavior of a Web application's real-world execution.

Stick Shift provides developers with several out-of-the-box instrumentation actions, including incrementing counters, capturing local variable state, capturing a return value, and reporting arbitrary log messages. In addition, developers can provide their own instrumentation code, written in JavaScript, for

Stick Shift to inject. With a few exceptions, a developer can inject these instrumentation actions into any of several kinds of instrumentation points within an application.

—*Function*. Instrumenting a function adds an instrumentation action to the entrance and exit of the function. By default, Stick Shift inserts actions to record function entry and exit, and to capture parameter and return values, as well as the local variable state at the end of the function execution.

—*Statement*. Instrumenting a statement adds an action either before or after a JavaScript statement.

—*Iterations*. Instrumenting an iteration, including `for` and `while` loops, inserts instrumentation action at the beginning or end of the loop.

—*Script block*. Scripts, script blocks, and event handlers can be instrumented at the entrance or exit to the script.

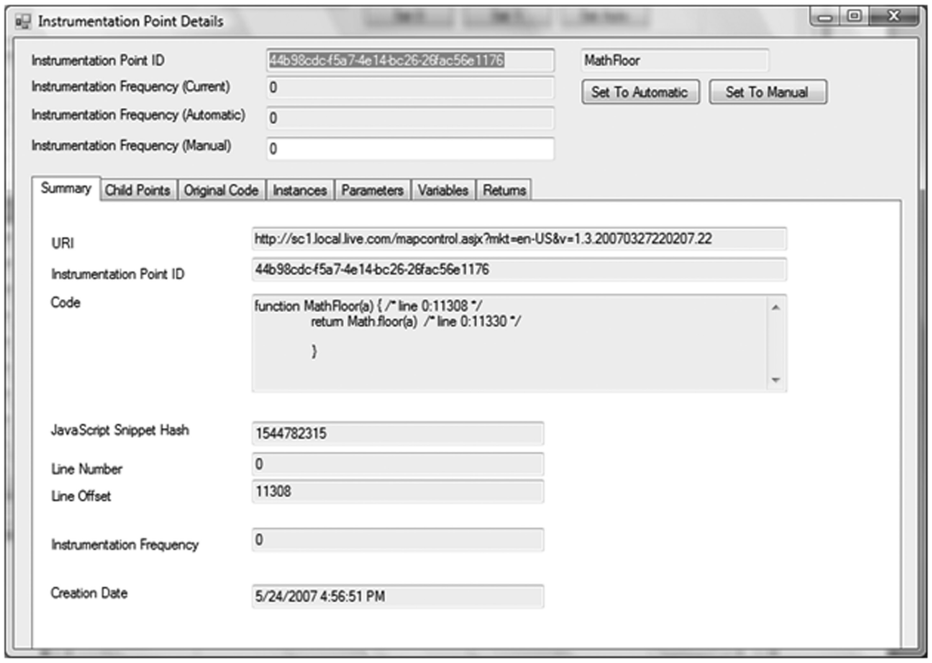
Figure 16 shows a screenshot of Stick Shift's display of a single function instrumentation point. The particular instrumentation point shown is a trivial `MathFloor(a)` function instrumented from a real-world Web application. The tool also provides an *instances* view, which displays the application instances (i.e., the Web sessions) which have executed this instrumentation point. A developer can use the instances view to drill down and display only the data captured from a single application instance. Other views display simple histograms of the parameters, variables, and return value data collected at this instrumentation point. These histograms are useful for manually discovering anomalous executions or otherwise exploring the usage behavior of an application.

9. DOLOTO

The use of client-side JavaScript code greatly improves the responsiveness of these network-bound applications. This shift of application execution from a back-end server to the client often dramatically increases the amount of code that must be downloaded to the browser. This creates an unfortunate Catch-22: to create responsive distributed Web 2.0 applications developers move code to the client, but for an application to be responsive, the code must first be transferred there, which takes time.

DOLOTO[7] is an optimization tool for Web 2.0 applications we have built on top of AjaxScope. With the help of JavaScript instrumentation, DOLOTO analyzes application workloads and *automatically* rewrites the existing application code to introduce dynamic code loading. After being processed by DOLOTO, an application will initially transfer only the portion of code necessary for application initialization. The rest of the application's code is replaced by short stubs—their actual implementations are transferred lazily in the background or, at the latest, on-demand on first execution of a particular application feature. Moreover, code that is rarely executed is rarely downloaded to the user browser. Because DOLOTO significantly speeds up the application startup

[7]DOLOTO stands for DOwnLOad Time Optimizer.



Instrumentation Point ID: 44b98cdc-f5a7-4e14-bc26-26fac56e1176 MathFloor
Instrumentation Frequency (Current): 0
Instrumentation Frequency (Automatic): 0
Instrumentation Frequency (Manual): 0

URI: http://sc1.local.live.com/mapcontrol.asjx?mkt=en-US&v=1.3.20070327220207.22
Instrumentation Point ID: 44b98cdc-f5a7-4e14-bc26-26fac56e1176
Code:

```
function MathFloor(a) { /* line 0:11308 */
          return Math.floor(a)  /* line 0:11330 */

          }
```

JavaScript Snippet Hash: 1544782315
Line Number: 0
Line Offset: 11308
Instrumentation Frequency: 0
Creation Date: 5/24/2007 4:56:51 PM

Fig. 16. Stick Shift screenshot.

and since subsequent code download is interleaved with application execution, applications rewritten with DOLOTO appear much more responsive to the end-user.

To demonstrate the effectiveness of DOLOTO in practice, we have performed experiments on five large widely used Web 2.0 applications. DOLOTO reduces the size of application code download by hundreds of kilobytes or as much as 50% of the original download size. The time to download and begin interacting with large applications is reduced by 20–40% depending on the application and wide-area network conditions. DOLOTO especially shines on wireless and mobile connections, which are becoming increasingly important in today's computing environments.

9.1 Overview of the Approach

The goal of DOLOTO is to automate the process of optimal code decomposition. DOLOTO's processing of application code automatically handles language issues such as closures and scoping; DOLOTO's analysis discovers an appropriate code decomposition for the likely execution order of functions. As a consequence of this sort of automation, developers no longer have to manually maintain the decomposed version of the application as the application or typical usage scenarios change, but simply reapply the analysis and decomposition as necessary.

DOLOTO's primarily targets feature-rich Web applications such as Live Maps. The code base of these applications is growing as they expand to provide functionality once reserved for traditional desktop software. Unfortunately, the latency cost of downloading this additional code is paid whether or not the additional features are used. This suggests that splitting the code of these features out, and dynamically loading them outside the critical-performance-path of initialization is likely to improve initial page loading times.

Overview. DOLOTO processing consists of two phases, the training and the execution phase described in the rest of this section. The training phase of DOLOTO's processing consists of running the application with its client-side JavaScript component instrumented to collect function-level profile information. The result of this training is an *access profile*, a clustering of original functions by time of their first use. In our implementation, training is performed by observing a user performing a fixed workload, although it is possible to train in a distributed manner, by combining workloads from multiple users with varying workloads, resulting in better code coverage and higher quality access profiles.

Access profiles may be either static, determined during initial training, or dynamic, being updated continuously and adjusting to operating conditions during application deployment. In DOLOTO, we present the profile information to the developer so that they can tweak the training parameters and review the resulting clusters. Notice that depending on how the application is executed, different cluster decompositions may make sense: for an application that is deployed on mobile devices, small clusters are appropriate, whereas going to the server to fetch a cluster consisting of only several kilobytes of code is probably wasteful for a Web application running on a desktop computer.

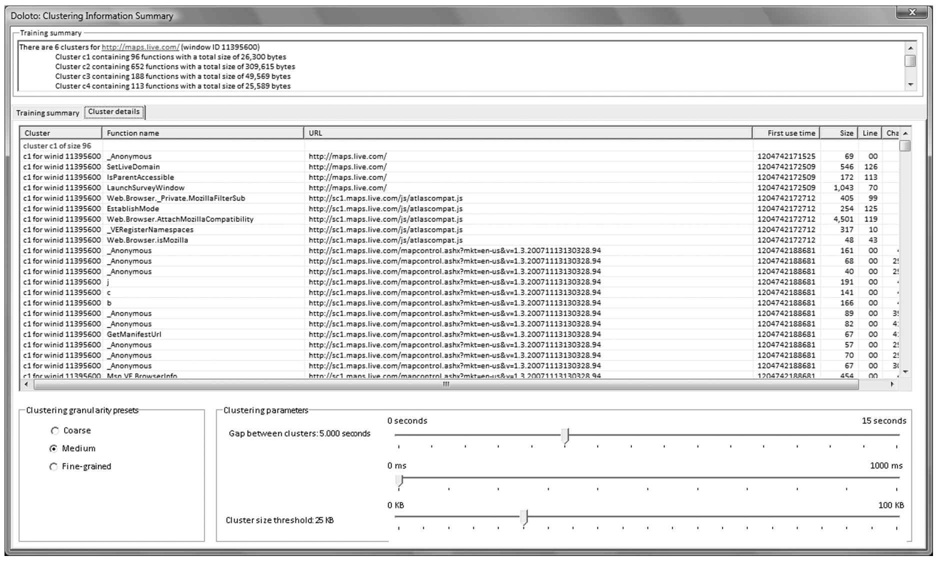
Collecting Access Profiles. At its core, our instrumentation approach is based on the ability to parse and instrument JavaScript code and to insert timestamps that allow us to group functions into clusters by the time of their first access. DOLOTO is an excellent example of how the AjaxScope platform can be used: the proxy-based instrumentation approach allows us to use a local proxy to obtain timing information for external, third-party sites.

The beginning of every function is instrumented to record the timestamp as well as the size of the function. At runtime, timestamps are collected by the instrumentation DOLOTO proxy and postprocessed to extract the first-access time ts_i for every function f_i that is observed at runtime. To avoid excessive network traffic, timestamp data is buffered on the client before being sent over to the training proxy. Because the exact code executed may vary depending on the browser version, we have built merging tools that combine multiple profiling runs obtained by training using a different browser.

The list of timestamps is sorted and traversed to group functions into clusters $c_1, \ldots, c_n$. As we are traversing the sorted list we are looking to terminate the current cluster c_j at function f_i according to the following criterion:

$$ts_{i+1} - ts_i > T_{gap} \wedge size(c_j) > T_{size},$$

That is, the time gap between the two subsequent functions exceeds the predefined gap threshold T_{gap} and the size of the current cluster exceeds the predefined size threshold T_{size}. Note that we disregard the original decomposition


Doloto: Clustering Information Summary

Training summary

There are 6 clusters for http://maps.live.com/ (window ID 11395600)
Cluster c1 containing 96 functions with a total size of 26,300 bytes
Cluster c2 containing 652 functions with a total size of 309,615 bytes
Cluster c3 containing 188 functions with a total size of 49,569 bytes
Cluster c4 containing 113 functions with a total size of 25,589 bytes

Training summary | Cluster details

Cluster	Function name	URL	First use time	Size	Line	Cha
cluster c1 of size 96						
c1 for winid 11395600	_Anonymous	http://maps.live.com/	1204742171525	69	00	
c1 for winid 11395600	SetLiveDomain	http://maps.live.com/	1204742172509	546	126	
c1 for winid 11395600	IsParentAccessible	http://maps.live.com/	1204742172509	172	113	
c1 for winid 11395600	LaunchSurveyWindow	http://maps.live.com/	1204742172509	1,043	70	
c1 for winid 11395600	Web.Browser._Private.MozillaFilterSub	http://sc1.maps.live.com/js/atlascompat.js	1204742172712	405	99	
c1 for winid 11395600	EstablishMode	http://sc1.maps.live.com/js/atlascompat.js	1204742172712	254	125	
c1 for winid 11395600	Web.Browser.AttachMozillaCompatibility	http://sc1.maps.live.com/js/atlascompat.js	1204742172712	4,501	119	
c1 for winid 11395600	_VERegisterNamespaces	http://sc1.maps.live.com/js/atlascompat.js	1204742172712	317	10	
c1 for winid 11395600	Web.Browser.isMozilla	http://sc1.maps.live.com/js/atlascompat.js	1204742172712	48	43	
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	161	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	68	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	40	00	[illegible]
c1 for winid 11395600	j	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	191	00	[illegible]
c1 for winid 11395600	c	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	141	00	[illegible]
c1 for winid 11395600	b	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	166	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	89	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	82	00	[illegible]
c1 for winid 11395600	GetManifestUrl	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	67	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	57	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	70	00	[illegible]
c1 for winid 11395600	_Anonymous	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	67	00	[illegible]
c1 for winid 11395600	Msn.VE.BrowserInfo	http://sc1.maps.live.com/mapcontrol.ashx?mkt=en-us&v=1.3.20071113130328.94	1204742188681	454	00	[illegible]

Clustering granularity presets

Coarse [✓ Medium] Fine-grained

Clustering parameters

Gap between clusters: 5.000 seconds (0 seconds – 15 seconds)

0 ms – 1000 ms

Cluster size threshold: 25 KB (0 KB – 100 KB)

Fig. 17. DOLOTO training tool that allows the developer to adjust clustering settings and training thresholds.

of functions into files: functions from different JavaScript files may and do end up belonging to the same cluster because of temporal proximity to each other.

Note that as with any runtime analysis, a potential weakness of this approach is that some code may not be used for the workload we apply: for instance, if the "help" functionality of an online mapping application is not utilized during the training run, functions implementing this functionality will be group into an special cluster $\perp$. As part of the process, map

$$\mathcal{P} : \{f_1, \ldots, f_k\} \rightarrow \{c_1, \ldots, c_n, \perp\}$$

from functions to clusters is saved as the access profile.

In practice, the cluster decomposition changes drastically depending on the threshold values. In our experiments below we favored threshold settings that produced about a dozen clusters that roughly correspond to high-level application activities. For instance, the activities of the initial page load, double-clicking on the map, moving the map around, asking for directions, printing the map for Live Maps may each be grouped in their own cluster.

However, for an application that is likely to run in a mobile setting where the user might be paying for bytes transferred, producing many clusters of a few kilobytes each is actually a good idea. It is also common to have slightly different versions of the same application for different browsers, so performing training for each browser separately, and then using the appropriate application version based on the execution environment is the right approach. Figure 17 shows how a DOLOTO screen presented to the user that allows him or her to adjust the thresholds, immediately seeing how that affects the number and composition of function clusters.

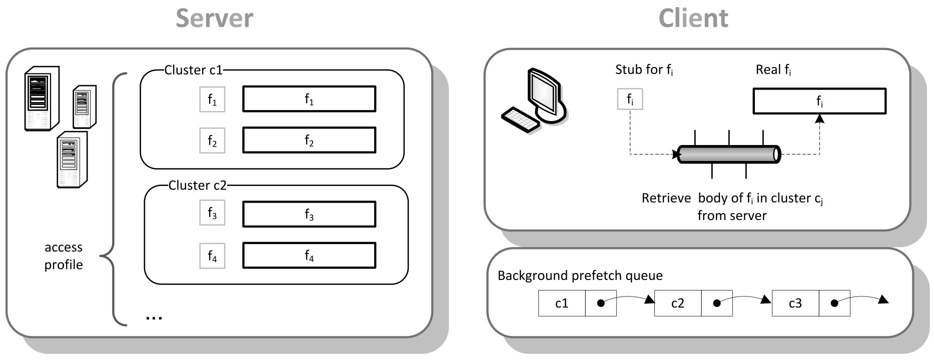


Fig. 18. Execution phase of DOLOTO: the left hand side of the picture shows code located on the server migrating to the client, shown on the right hand side. The execution phase relies on profiles computed during the training phase.

Examples above illustrate that there is no "perfect" access profile and that the access profile should be customized based on how the application is likely to be used. Moreover, a small benchmark can be injected into the beginning of the application run to determine the network and CPU conditions for a particular user; this information can be cached on the client for subsequent application runs. Based on this data, the application will proceed to download one of several application versions. However, unlike the current practice of developing a separate "mobile" version of an application, all versions would be based on the same code. The execution phase of DOLOTO is illustrated in Figure 18.

Code rewriting. The basis for our approach is to rewrite every JavaScript function `f(x)` with a stub that in its simplified version looks as follows:

```
function f(x){
    var real_f_text = blocking_download("f");
    var real_f_func = eval(real_f_text);
    return real_f_func.apply(this, arguments);
}
```

Where `blocking_download` is a synchronous call that retrieves the body of function `f` from the server. A network call is made only once per cluster: if the body of `f` has already been transferred to the client either on-demand or background code loading, `blocking_download` returns it immediately. We proceed to `eval` the body at runtime and apply the resulting function to the arguments that are being passed into `f`. For details of rewriting we refer the reader to Livshits and Kıcıman [2008].

9.2 Experimental Setup

The goal of our experiments is to evaluate the impact of code splitting on the download size and initial responsiveness of real-world, third-party Web applications for a variety of realistic network conditions. In the setup of our experimental testbed, we face several challenges:

Modifying third-party applications. While in our design, DOLOTO is intended to be deployed as part of the server-side infrastructure for Web applications, we do not, in our testbed, have control over the server-side environments of the applications with which we are experimenting. In order to apply DOLOTO to these applications, we implement DOLOTO as a rewriting proxy that intercepts the responses from third-party Web servers and dynamically rewrites their JavaScript content using our code splitting policies. In all our experiments, our client-side Web browsers are chained to the proxy implementation of DOLOTO. To accurately simulate a server-side deployment of DOLOTO with offline rewriting of application code, we ensure that our dynamic rewriting is not in the critical path of serving Web pages. Thus, our DOLOTO proxy caches the results of its rewritings such that a second visit to the page is immediately fulfilled.

Sites serving multiple versions of Web application code. Web applications frequently serve different versions of their code over time, either as part of a rolling upgrade or as part of a concurrent A/B test of new functionality. To get comparable and consistent results, our experiments rely on training on and executing the same Web application code. To be sure that our experiments are always run against the same version, we deploy the Squid caching proxy [Squid Developers 2006] to hold a single copy of our benchmark Web application code. To ensure that all components of the Web applications are cached, we use an additional HTTP rewriting proxy, Fiddler [Lawrence 2007], that forces all components of a Web application to be cache-able by modifying the HTTP cache-control headers set by the original Web site.

Simulating realistic network conditions. In order to collect execution times for a realistic range of network conditions, we use a wide-area network simulator that provides control over the effective bandwidth, latency, and packet loss rates of a machine's network connection. We use this network simulator to simulate three different environments: a Cable/DSL connection with a low-latency network path to a Web site (300 kbps downstream bandwidth and 50 msec round-trip latency); a Cable/DSL connection with a high-latency network path (300 kbps downstream bandwidth and 300 msec round-trip latency); and a 56k dial-up connection (50 kbps downstream bandwidth and 300 msec round-trip latency).

Our training setup uses Fiddler, Squid, and DOLOTO. The DOLOTO proxy is running on a machine with a dual Intel Xeon, 3.4GHz CPU, with 2.5GB of RAM. The client-side machine is a Pentium 4 3.6 GHz machine equipped with 3 GB of memory running Windows Vista with Firefox 2.0 as the browser. The physical network connection between all our test machines is a 100 Mb local area network over a single hub.

In our testing setup, we first populate a Squid cache running a workload through DOLOTO so that the DOLOTO-processed version of the application is saved in the cache. We then put a a wide-area network simulator between the Squid cache and the browser to evaluate a range of network conditions and replay the same application workload.

9.3 Experimental Results

This section provides a summary of experimental results for DOLOTO; more results are given in Livshits and Kıcıman [2008]. Below we talk about DOLOTO training and execution phase statistics.

Training Phase Statistics. To train the clusters and create the access profiles for a Web application, we collected a profile of several minutes of each Web application's execution under a manual workload that exercised a variety of each application's functionality. For example, the manual workload for Bunny Hunt and the Chi game consists of playing the game, and the workload for maps.live.com consists of browsing and searching through the map.

A summary of results for the training phase is shown in Table VIII. Column 2 shows the total (uncompressed) download size for each application in our benchmark suite. Columns 3–6 show information about the code coverage observed during our training run, detailing the number of functions called during the run (absolute number and percentage in columns 3–4) and the *size* of these functions (absolute number and percentage in columns 5–6). Columns 7–10 show a distribution of function sizes that we have observed at runtime. While small functions are quite common, especially in obfuscated sites that deliberately introduce them, there are quite a number of large functions as well, indicating the potential to benefit from removing functions from the initial download.

Finally, columns 11–13 show information about the clusters we constructed. Column 11 shows the minimum-average-maximum number of functions per cluster. As can be seen from the table, it is fairly typical to have a dozen clusters for the larger applications, with some clusters being quite sizable, containing several hundred functions and tens of kilobytes of code in the case of larger applications. Also, it is quite common for a cluster to contain functions from more than one file. This demonstrates our reliance on realistic workloads to recompose the code instead of the initial code decomposition provided by the developer. Column 13 shows the average cluster size, in KB.

Note that the number of clusters is quite sensitive to the threshold selection. For these results, we used a threshold of 25 msec for the gap between first-run times for functions and a minimum cluster size threshold of 1.5 KB. We created at least one cluster per frame for applications that contained multiple `FRAME` or `IFRAME` tags. For the purposes of measuring download size and time improvements, we ensured that all the functions used during page initialization were included in the initial cluster together.

Execution Phase Statistics. Table IX shows the reduction of size achieved with DOLOTO rewriting for our application benchmarks. Columns 2 and 3 show information about the number of percentage of the functions that are rewritten to insert stubs. Since the first cluster is not rewritten and is pushed to the application verbatim, less than 100% of all functions end up being stubbed. Columns 4–6 show the size of the regular (uncompressed) code in its original version, the size of the initial DOLOTO download that includes all the stubs that are sent to the client initially, and the resulting space savings. Columns 7–10

Table VIII. Training Statistics for Our Benchmark Applications

		Code coverage in training				**Function characterization**				**Cluster statistics**		
						Size distribution (characters)						
Web application	**Download size, in KB**	**Number functions**	**%**	**Total size, in KB**	**%**	**<100**	**100-200**	**200-500**	**>500**	**Number of clusters**	**Functions per cluster**	**Average size, in KB**
Chi game	104	103	29%	43	41%	22	26	28	27	7	3/14/43	6.2
Bunny Hunt	16	22	44%	10	60%	5	0	9	8	3	2/7/19	3.3
Live.com	1,436	689	21%	572	39%	203	149	165	172	14	5/49/461	40.9
Live Maps	1,909	803	16%	835	43%	284	188	177	154	12	6/66/689	69.7
Google Spreadsheets	499	794	24%	179	35%	442	156	121	75	15	3/52/648	12.0

Table IX. Size Reduction After DOLOTO Rewriting for Our Benchmarks

			Application size, in bytes								
Web	**Functions**		**Regular**			**Crunched**			**Crunched and Gzipped**		
application	**rewritten**		**Original**	DOLOTO	**Savings**	**Original**	DOLOTO	**Savings**	**Original**	DOLOTO	**Savings**
Chi game	202	71%	125,045	69,469	44%	124,647	69,073	45%	34,227	21,004	39%
Bunny Hunt	24	61%	17,371	7,841	55%	17,216	7,692	55%	4,836	3,033	37%
Live.com	1,680	58%	882,438	472,706	46%	882,409	472,680	46%	220,544	129,278	41%
Live Maps	1,463	42%	1,125,618	617,183	45%	1,125,600	617,171	45%	270,644	155,992	42%
Google Spreadsheets	1,382	48%	654,192	402,060	38%	654,142	402,010	38%	180,367	96,452	46%

Table X. Reduction in Execution Times Achieved with DOLOTO for Different Connection Parameters. For Each Group of Columns, the Table Shows the Original Download Time and the Time with DOLOTO. Both Times are Measured in Seconds and the "%" Column Shows the Time Savings as a Percentage

Web	**50kbs/300ms**			**300kbs/300ms**			**300kbs/50ms**		
application	**Original**	**Doloto**	%	**Original**	**Doloto**	%	**Original**	**Doloto**	%
Chi game	37	37	0	13	15	13	8	8	0
Bunny Hunt	100	92	8	43	41	5	22	22	0
Live.com	99	82	17	31	28	10	18	13	28
Live Maps	155	112	28	31	23	26	26	19	27
Google Sp'sheet	58	45	22	20	20	0	18	11	39

show the same numbers with the code having been run through a JavaScript crunching utility that removes superfluous whitespace—a common strategy for optimizing released JavaScript code. The tool configuration we used did not perform any additional optimizations such as shortening local variable identifiers. Finally, columns 11–14 show the same set of numbers after the code has been crunched *and* run through a gzip compression utility.

In today's AJAX applications, gzip compression is a common and perhaps the easiest strategy for reducing the amount of data transferred over the network, and it is used widely by the sites we chose as our benchmarks. Other compression standards are likely to emerge in the future. Because DOLOTO transfers well-formed portions of the program AST, we note that DOLOTO integrates well with size-saving approaches based on AST compression [Burtscher et al. 2010].

In addition to size reduction measurements, we also performed detailed experiments with several representative benchmarks to determine the effect of code size reduction on the overall application execution time for a range of network parameters, as shown in Table X. For each group of columns in Table X, we show the original execution time, the time with DOLOTO, and the percentage of time savings. Clearly, whether the size reduction is accomplished by DOLOTO will translate into execution time reduction depends heavily on application decomposition. For instance, it is not uncommon to have images and JavaScript code as the biggest application components; below we consider applications with different ratios of the two.

It is not too surprising that, as an application whose download is dominated by images, Bunny Hunt does not show any significant improvements with DOLOTO. On the other hand, mash-up site Live.com, which has JavaScript as its most significant download component, shows pretty significant speedups, especially in the case of a low-latency high-bandwidth connection. For high-latency connections, the time savings are tangible, but not as significant because the execution time is dominated by the need to connect to many servers to fetch data to be shown on the mash-up page.

Live Maps shows 26-28% improvements for a range of network conditions, with dozens of seconds being saved on the slowest connection. This is quite impressive given that a significant portion of the application execution is spent on retrieving map images. However, as Table IX shows, these time savings can be explained by the fact that about 45% of the application code is not being transferred in the DOLOTO version. Time savings are most significant for Google Spreadsheet, in which code is the most significant download component.

Because the entire application is under 200 KB in size and the image component is quite small, savings accomplished with DOLOTO result in noticeable speedups. However, on a 300 msec latency connection, third-party server requests that are used for analytics collection dominate the download time, masking the savings achieved with DOLOTO.

9.4 Summary

We believe that a dynamic code loading technology like DOLOTO goes a long way towards enabling very large AJAX applications. When we published the DOLOTO paper, dynamic loading techniques in the context of AJAX were considered exotic; these days, they are becoming commonplace for large, complex JavaScript applications. As we mentioned before, this is driven by the size of the JavaScript code that needs to be downloaded to the client for rich AJAX applications. We have seen popular libraries such as jQuery and translators such as Script# adapting many of the ideas of DOLOTO to perform dynamic code loading. We see large commercial applications being restructured to support modular code loading. Moreover, we think that going forward, DOLOTO technology can be made even better with enhanced browser support for code caching and loading.

10. ADDITIONAL AJAXSCOPE POLICIES

To provide a sense of the range of useful instrumentation policies AjaxScope can be used for, in this section we detail policies that we have not yet implemented.

10.1 Data Structure Corruption Bugs

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 written by a developer or automatically inferred [Demsky et al. 2006]. For example, a doubly-linked list data structure might be inspected for unmatched forward and backward references.

When a consistency check fails, we might suspect that a bug exists somewhere in the executed code after the last successful consistency check.[8] 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.

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

With AjaxScope 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.

10.2 Monitoring Third-Party Services

AjaxScope gives the developer visibility into their code's performance at different levels of granularity. While function-level profiling described in Section 5.3 may be useful in detecting first-order performance bottlenecks, developers are often interested in the performance of specific portions of code.

For instance, the developer of a dynamically assembled page that loads third-party advertisements together with static text may be interested in the performance of ad loading. If ad loading is perceived as sluggish, users may be turned off and stop using the service. For graphical ads, images are loaded asynchronously, so the naïve strategy of injecting a piece of JavaScript to record the timestamp *after* the image loading code does not work. Instead, we can use AjaxScope to inject a piece of JavaScript that does polling, like the one shown in the following.

```
var reportedAlready = false;
function checkImages(){
    if(reportedAlready) return;
    for(var i =0; i <  images.length; i++){
        var img = images[i];
        if(!img.complete) return;
    }
    ajaxScopeSend('Finished loading images at '
                    + ajaxScopeTimestamp());
        reportedAlready = true;
    }
    setTimeout('checkImages()', 100);
```

This code traverses the array of images checking for whether their loading has finished every .1 second. When it is, the timestamp is sent to the server.

10.3 Monitoring Wide-Area, End-to-End Network Performance

AjaxScope provides visibility into the network overhead as perceived within users' browser. The network overhead can be quite severe for Web application users located on a different continent. However, is is difficult to assess it in a local testing environment. Since AjaxScope records user's IP addresses, it is possible to correlate overhead numbers with users' geography. AjaxScope allows us to instrument `XmlHttpRequest` calls to record start and finish times.

Example 6. A typical `XmlHttpRequest` post to the server may look like this

```
xhr = new XMLHttpRequest();
xhr.onreadystatechange  = function {
    // process server response
}
xhr.open(GET, "http://www.cnn.com/news.xml",  true);
xhr.send(null);
```

With AjaxScope we can rewrite the `send` call to capture the initial timestamp and also the entry of the `onreadystatechange` handler to record the completion timestamp. The difference between the two gives us an estimate of the network round-trip times.

11. DISCUSSION

Section 11.1 presents possible deployment scenarios for AjaxScope. Section 11.2 addresses potential reliability risks involved in deploying buggy instrumentation policies. Issues of privacy and security that might arise when extra code is executing on the client-side are addressed in Section 11.3. Finally, Section 11.4 addresses the interaction of AjaxScope and browser caching.

11.1 AjaxScope Deployment Scenarios

The AjaxScope proxy can be deployed in a variety of settings. While client-side deployment is perhaps the easiest, we envision AjaxScope deployed primarily on the server side, in front of a Web application or a suite of applications. In the context of load balancing, which is how most widely used sites today are structured, the functionality of AjaxScope can be similarly distributed in order to reduce the parsing and rewriting latency. Server-side deployment also allows developers or system operators to tweak the "knobs" exposed by individual AjaxScope policies. For instance, low-overhead policies may always be enabled. Others higher-overhead policies may only be turned on on-demand, after a change that is likely to compromise system reliability, such as a major system update or a transition to a new set of APIs.

AjaxScope can be used by Web application testers without necessarily requiring support from the development organization, as demonstrated by our experiments with third-party code. AjaxScope can also be used in a test setting when it is necessary to obtain detailed information from a single user. Consider a performance problem with Hotmail, which only affects a small group of users. With AjaxScope, when a user complains about performance issues, she may be told to redirect her browser to an AjaxScope proxy deployed on the server side. The instrumentation performed can also be customized depending on the bug report. That way, she will be the only one running a specially instrumented version of the application, and application developers will be able to observe the application under the problematic workload. A particularly attractive feature of this approach is that no additional software needs to be installed on the client side. Moreover, real-life user workloads can be captured with AjaxScope for future use in regression testing. This way real-life workloads can be used, as opposed to custom-developed testing scripts.

AjaxScope also makes gradual proxy deployment quite easy: there is no need to install AjaxScope on all servers supporting a large Web application. Initially, a small fraction of them may be involved in AjaxScope deployment. Alternatively, only a small fraction of users may initially be exposed to AjaxScope.

Our article does not explore the issues of large-scale data processing, such as data representation and compression as well as various ways to present and visualize the data for system operators. For instance, network overhead can be measured and superimposed onto a map in real time. This way, when the performance of a certain region, as represented by a set of IP addresses, goes down, additional instrumentation can be injected for only users within that IP range to investigate the reason for the performance drop.

11.2 Policy Deployment Risks

Users appreciate applications that have predictable behavior, so we do not want to allow policies to significantly impact performance, introduce new errors, etc. New policies can also be deployed in a manner that reduces the chances of negatively affecting application users. After the application developers have debugged their instrumentation, more users can be redirected to AjaxScope. To ensure that arbitrary policies do not adversely affect predictability, our infrastructure monitors every application's coarse-grained performance and observed error rate. Monitoring is done via a trusted instrumentation policy that makes minimal changes to application code, an approach we refer to as *metamonitoring*.

When a buggy policy is mistakenly released and applied to a Web application, some relatively small number of users will be affected before the policy is disabled. This meta-monitoring strategy is not intended to make writing broken policies acceptable. Rather, it is intended as a backup strategy to regular testing processes to ensure that broken policies do not affect more than a small number of users for a short period of time.

11.3 Information Protection

Existence of the JavaScript sandbox within the browser precludes security concerns that involve file or process manipulation. We argue that AjaxScope does not weaken the security posture of an existing Web application, as there is already a trust relationship between a user and a Web application. Moreover, a level of protection is achieved through the use of the browser's sandbox. However, one corner-case occurs when Web applications wish to carefully silo sensitive information. For example, e-commerce and financial sites carefully engineer their systems to ensure that critical personal information, such as credit card numbers, are stored only on trusted, secured portions of their data centers. Arbitrary logging of information on the client can result in this private information making its way into a comparatively insecure logging infrastructure.

One option to deal with this is to add dynamic information tainting [Nguyen-Tuong et al. 2005; Wall et al. 1996; Martin et al. 2006], which can be easily done using our rewriting infrastructure. In this case, the Web application developer would cooperate to label any sensitive data, such as credit card numbers. The running instrumentation policies would then refuse to report the value of any tainted data.

11.4 Caching Considerations

Today, almost all Web applications use client-side caching to achieve a faster user experience. These Web applications, built using a collection of JavaScript code files, transfer their code to the client the first time a Web application is accessed, and then keep these files cached at the client to improve performance during subsequent visits. Instant redeployment, critical to many key AjaxScope features, however, requires that clients check for new versions of code with the server, conflicting with the benefits of caching. The challenge then is to achieve the benefits of distributed and adaptive instrumentation without compromising the performance benefits of caching.

Distributed Instrumentation. A simple strategy for achieving distributed instrumentation in the context of client-side caching is straightforward: AjaxScope can continue to serve differently instrumented files and allow them to be cached as before. Caching on clients will have no affect on the distribution of these files, while proxies that cache files for many clients will skew the sampling. Regardless of proxy caches, a large service will still achieve coverage of all the desired instrumentation points.

Adaptive Instrumentation. Achieving adaptive instrumentation in the context of client-side caching is more difficult. There are two negative side effects when clients cache adaptively instrumented JavaScript code. First, the caching clients may no longer be providing relevant instrumentation data—even if AjaxScope has learned enough from the instrumentation cached by the client, it will not be able to update the client until its cache expires or is revoked. Secondly, the client may be running a particularly heavy weight instrumentation that was not intended to be in use for long, and thus paying a runtime performance penalty. As long as there are sufficient clients accessing the Web application with either empty or expired caches, the first side effect AjaxScope will be able to find new sources of data as it adapts its instrumentation. The affect of the second issue on user-perceived performance is potentially greater. In this case, we recommend that AjaxScope set short expiry times on any heavyweight instrumentation.

In both of these cases, AjaxScope can take advantage of existing techniques for reducing the burden on Web clients of ensuring the freshness of cached files. One such technique is to combine a noncacheable top-level HTML page with long-lived dependent resources with version information embedded in their identifier. In this case, the dependent resources are never directly updated (e.g., during reinstrumentation). Instead, the top-level HTML page is updated to reference a new version of the dependent resource. Using this technique, the client does not need to check the validity of every cached object, and in the common case does not need to re-download large, dependent JavaScript, image, or CSS files.

12. RELATED WORK

Several previous projects have worked on improved monitoring techniques for Web services and other distributed systems [Barham et al. 2004; Aguilera et al. 2003], but to our knowledge, AjaxScope is the first to extend the developer's visibility into Web application behavior onto the end-user's desktop. Other researchers, including Tucek [Tucek et al. 2006], 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 addition, much recent work has been done on exploiting peer-to-peer or client-to-server relationships to monitor and validate the correctness of nodes in a distributed system [Haeberlen et al. 2007; Michalakis et al. 2007; Yumerefendi and Chase 2007].

12.1 Dynamic Instrumentation

While program instrumentation has been used for desktop application development for a very long time, we feel that AjaxScope is a novel contribution. Unlike much prior work, AjaxScope allows developers to gain insight into application behavior in a wide-area setting across administrative domains. Perhaps the closest in spirit to our work is ParaDyn [Miller et al. 1995], which uses dynamic, adaptive instrumentation to find performance bottlenecks in parallel computing applications. Sun's DTrace [Microsystems 2009] also provides a form of adaptive instrumentation through dynamic interposition for monitoring of the operating system and user programs running on an individual machine. Sirer et al. demonstrated the use of proxy-based rewriting of network-deployed applications in Sirer et al. [1999], using this technique to enforce security policies.

12.2 Runtime Analysis

Much research has been done in runtime analysis for finding optimization opportunities [Rubin et al. 2002; Chilimbi and Shaham 2006; Martin et al. 2005]. In many settings, static analysis is used to remove instrumentation points, leading to a reduction in runtime overhead [Martin et al. 2005]. However, the presence of the `eval` statement in JavaScript as well as the lack of static typing make it a challenging language for analysis. Moreover, not the entire code is available at the time of analysis. However, we do believe that some instrumentation policies can definitely benefit from static analysis, which makes it a promising future research direction.

In recent years, runtime program analysis has emerged as a powerful tool for finding bugs, ranging from memory errors [Rinard et al. 2004; Cowan et al. 1998; Berger and Zorn 2006; Hauswirth and Chilimbi 2004; Chilimbi and Shaham 2006] to security vulnerabilities [Haldar et al. 2005; Martin et al. 2006; Nguyen-Tuong et al. 2005]. An area of runtime analysis that we believe to be closest to our work is statistical debugging. Statistical debugging uses runtime observations to perform bug isolation by using randomly sampled predicates of program behavior from a large user base [Liblit et al. 2005; Liu et al. 2006; Liu and Han 2006]. We believe that the adaptive instrumentation of AjaxScope can improve on such algorithms by enabling the use of active learning techniques [Cohn et al. 1996].

12.3 JavaScript Code Rewriting

Both BrowserShield and CoreScript use JavaScript rewriting to enforce browser security and safety properties [Reis et al. 2006; Yu et al. 2007]. AjaxScope's focus on nonmalicious scenarios, such as developers debugging their own code, allows us to simplify our rewriting requirements and make different trade-offs to improve the performance and simplicity of our architecture. For example, BrowserShield implements a JavaScript parser in JavaScript and executes this parser in the client browser to protect against potentially malicious, runtime generated code. In contrast, our parser executes in a proxy and any dynamically generated code is either not instrumented or must be sent back to the proxy to be instrumented.

Much of the focus in JavaScript code rewriting has been on security. Caja [Miller et al. 2007] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [Microsoft Live Labs 2008]. While we are not aware of a published comprehensive overhead evaluation, it appears that the runtime overhead of Caja and WebSandbox may be high, depending on the level of rewriting. This overhead comes from the need to fully mediate and inspect access to just about any field reference. For instance, a Caja authors' report suggest that the overhead of various subsets that are part of Caja are 6–40x [Miller 2009]. The focus of AjaxScope is generally on less aggressive and less costly rewriting.

12.4 Static Analysis of JavaScript

Approaches to analyzing JavaScript statically so far have included type inference and pointer analysis; both have been the subject of active research in the last several years.

It has often been observed that a more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [Cartwright and Fagan 2004] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety. Other work has been done to devise a static type system that describes the JavaScript language [Anderson et al. 2005; Anderson and Giannini 2004; Thiemann 2005; Jensen et al. 2009]. These typically works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets of JavaScript. However, analysis of large-scale JavaScript applications that use difficult-to-analyze features such as `eval` remains largely out of reach.

Gatekeeper and Gulfstream [Guarnieri and Livshits 2009] uses a pointer analysis to reason about JavaScript programs. The ability to reason about pointers and the program call graph often allows one to express more interesting security policies than would be possible otherwise. A recent project by Chugh et al. [2009] focuses on staged analysis of JavaScript and finding information flow violations in client-side code. Chugh et al. focus on information flow properties such as reading document cookies and changing the locations. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. We should point out that AjaxScope does not suffer from the requirement to have the entire program available for instrumentation at once. The topic of streaming JavaScript applications and incremental analysis is further explored in the Gulfstream project [Guarnieri and Livshits 2010].

13. CONCLUSIONS

In this article we have presented AjaxScope, a platform for improving developer's end-to-end visibility into Web application behavior through a continuous, adaptive loop of instrumentation, observation, and analysis. We have demonstrated the effectiveness of AjaxScope by implementing a variety of practical instrumentation policies for debugging and monitoring Web applications, including performance profiling, memory leak detection, and cache placement for expensive, deterministic function calls. We have applied these policies to a suite of 90 widely used and diverse Web applications to show that 1) adaptive instrumentation can reduce both the CPU overhead and network bandwidth, sometimes by as much as 30% and 99%, respectively; and 2) distributed tests allow us fine-grained control over the execution and network overhead of otherwise prohibitively expensive runtime analyses.

While our article has focused on JavaScript rewriting in the context of Web 2.0 applications, we believe that we have just scratched the surface when it comes to exploiting the power of instant redeployment for software-as-a-service applications. In the future, as the software-as-a-service paradigm, centralized software management tools [Chandra et al. 2005] and the property of instant redeployability become more widespread, AjaxScope's monitoring techniques have the potential to be applicable to a broader domain of software. Moreover, the implications of instant redeployability go far beyond simple execution monitoring, to include distributed user-driven testing, distributed debugging, and potentially adaptive recovery techniques, so that errors in one user's execution can be immediately applied to help mitigate potential issues affecting other users.

ACKNOWLEDGMENTS

We greatly appreciate the detailed comments and feedback of Andrew Myers, our SOSP shephard, our anonymous SOSP reviewers, and our anonymous reviewers, from *ACM Transactions on the Web*. Our discussions with Helen Wang, Trishul Chilimbi, Yi-Min Wang were invaluable to the conception and refinement of the project. We thank Chen Ding for his help formulating the initial ideas for Doloto [Livshits and Ding 2007]. We are grateful to our interns, Himanshu Sharma, who wrote an initial JavaScript-based instrumentation engine, and Troy Ronda, who built the MEDic extensions to AjaxScope. We would like to thank Hesham Anan, Shobana Balakrishnan, Brian Crawford, John Cunningham, Adam Eversole, and Eric Lawrence for all of their help in making AjaxScope more applicable to real-world scenarios.

REFERENCES

AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. 2003. Performance debugging for distributed systems of black boxes. In *Proceedings of the Symposium on Operating Systems Principles*. 74–89.

ANDERSON, C. AND GIANNINI, P. 2004. Type checking for JavaScript. In *Proceedings of the 2nd Workshop on Object-Oriented Development*. http://www.binarylord.com/work/js0wood.pdf.

ANDERSON, C., GIANNINI, P., AND DROSSOPOULOU, S. 2005. Towards type inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*. 429–452.

ATTERER, R., WNUK, M., AND SCHMIDT, A. 2006. Knowing the user's every move: user activity tracking for Website usability evaluation and implicit interaction. In *Proceedings of the International Conference on World Wide Web*. 203–212.

BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. 2004. Using Magpie for request extraction and workload modelling. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 259–272.

BARON, D. 2001. Finding leaks in Mozilla.
http://www.mozilla.org/performance/leak-brownbag.html.

BERGER, E. D. AND ZORN, B. G. 2006. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN Notes 41*, 6, 158–168.

BOSWORTH, A. 2006. How to provide a Web API.
http://www.sourcelabs.com/blogs/ajb/2006/08/how_to_provide_a_Web_api.html.

BREEN, R. 2007. Ajax performance. http://www.ajaxperformance.com.

BRUTLAG, J. 2009. Speed matters for google Web search.
http://code.google.com/speed/files/delayexp.pdf.

BURTSCHER, M., LIVSHITS, B., SINHA, G., AND ZORN, B. G. 2010. Jszap: Compressing JavaScript code. In *Proceedings of the USENIX Conference on Web Application Development*.

CARTWRIGHT, R. AND FAGAN, M. 2004. Soft typing. *ACM SIGPLAN Notices 39*, 4, 412–428.

CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. 2005. The Collective: A cache-based system management architecture. In *Proceedings of the Symposium on Networked Systems Design and Implementation*.

CHILIMBI, T. M. AND SHAHAM, R. 2006. Cache-conscious coallocation of hot data streams. *SIGPLAN Notes 41*, 6, 252–262.

CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. 2009. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*.

COHN, D. A., GHAHRAMANI, Z., AND JORDAN, M. I. 1996. Active learning with statistical models. *J. Artif. Intelli. Resear. 4*, 129–145.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Conference*. 63–78.

CRISP. 2006. String performance in Internet Explorer.
http://therealcrisp.xs4all.nl/blog/2006/12/09/string-performance-in-internet-explorer/.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, 205–220.

DEMSKY, B., ERNST, M., GUO, P., MCCAMANT, S., PERKINS, J., AND RINARD, M. 2006. Inference and enforcement of data structure consistency specifications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

ECMA. 1999. ECMAScript Language Specification 3rd Ed.
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

GUARNIERI, S. AND LIVSHITS, B. 2009. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the Usenix Security Symposium*.

GUARNIERI, S. AND LIVSHITS, B. 2010. Gulfstream: Incremental static analysis for streaming JavaScript applications. In *Proceedings of the USENIX Conference on Web Application Development*.

HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. 2007. Peerreview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, 175–188.

HALDAR, V., CHANDRA, D., AND FRANZ, M. 2005. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*. 303–311.

HAUSWIRTH, M. AND CHILIMBI, T. M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 156–164.

INTERNET EXPLORER DEVELOPMENT TEAM. IE+JavaScript performance recommendations part 2: JavaScript code inefficiencies.
http://therealcrisp.xs4all.nl/blog/2006/12/09/string-performance-in-internet-explorer/.

JENSEN, S. H., MØLLER, A., AND THIEMANN, P. 2009. Type analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS'09)*. Lecture Notes in Computer Science, vol. 5673. Springer-Verlag.

LAWRENCE, E. 2007. Fiddler: Web debugging proxy. http://www.fiddlertool.com/fiddler/.

LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. 2005. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*. 15–26.

LIU, C., FEI, L., YAN, X., HAN, J., AND MIDKIFF, S. P. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Engin. 32*, 10, 831–848.

LIU, C. AND HAN, J. 2006. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 46–56.

LIVSHITS, B. AND DING, C. 2007. Code splitting for network bound Web 2.0 applications. Tech. rep., Microsoft Research.

LIVSHITS, B. AND KICIMAN, E. 2008. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*.

MARTIN, M., LIVSHITS, B., AND LAM, M. S. 2005. Finding application errors and security vulnerabilities using PQL: A program query language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

MARTIN, M., LIVSHITS, B., AND LAM, M. S. 2006. SecuriFly: Runtime vulnerability protection for Web applications. Tech. rep., Stanford University.

MEYEROVICH, L. AND LIVSHITS, B. 2010. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the IEEE Symposium on Security and Privacy*.

MICHALAKIS, N., SOULE, R., AND GRIMM, R. 2007. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*. 145–158.

MICROSOFT LIVE LABS. 2008. Live Labs Websandbox. http://Websandbox.org.

MICROSYSTEMS, S. 2009. Dtrace. http://www.sun.com/bigadmin/content/dtrace/index.jsp.

MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. 1995. The ParaDyn parallel performance measurement tool. *IEEE Comput. 28*, 11, 37–46.

MILLER, M. S. 2009. Is it possible to mix ExtJS and google-caja to enhance security. http://extjs.com/forum/showthread.php?p=268731#post268731.

MILLER, M. S., SAMUEL, M., LAURIE, B., AWAD, I., AND STAY, M. 2007. Caja: Safe active content in sanitized JavaScript.
http://google-caja.googlecode.com/files/caja-2007.pdf.

NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J., AND EVANS, D. 2005. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*.

REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. 2006. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

RIDER, S. 2005. Recent changes that may break your gadgets.
http://microsoftgadgets.com/forums/1438/ShowPost.aspx.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. 2004. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 303–316.

RUBIN, S., BODIK, R., AND CHILIMBI, T. 2002. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Notes 37*, 1, 140–153.

SCHLUETER, I. Z. 2006. Memory leaks in Microsoft Internet Explorer.
http://isaacschlueter.com/2006/10/msie-memory-leaks/.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2002. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the International Symposium on Memory Management*. 64–75.

SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. 1999. Design and implementation of a distributed virtual machine for networked computers. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, New York, 202–216.

SQUID DEVELOPERS. 2006. Squid Web proxy cache. http://www.squid-cache.org.

THIEMANN, P. 2005. Towards a type system for analyzing JavaScript programs. In *Proceedings of the European Symposium on Programming*.

TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. 2006. Automatic on-line failure diagnosis at the end-user site. In *Proceedings of the Workshop on Hot Topics in System Dependability*.

WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. 1996. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA.

YU, D., CHANDER, A., ISLAM, N., AND SERIKOV, I. 2007. JavaScript instrumentation for browser security. In *Proceedings of the Symposium on Principles of Programming Languages*. 237–249.

YUE, C. AND WANG, H. 2009. Characterizing insecure JavaScript practices on the Web. In *Proceedings of the International World Wide Web Conference*.

YUMEREFENDI, A. R. AND CHASE, J. S. 2007. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. 77–92.

ZAKAS, N. C., MCPEAK, J., AND FAWCETT, J. 2006. *Professional Ajax*. Wrox.

Received September 2008; revised April 2010; accepted June 2010