

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

Emre Kiciman and Benjamin Livshits

Microsoft Research
Redmond, WA, USA

{emrek, livshits}@microsoft.com

ABSTRACT

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 paper 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 large amounts of JavaScript.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Distributed debugging; D.4.7 [Distributed systems]: Organization and Design

General Terms

Reliability, Performance, Measurement, Management, Languages

Keywords

Web applications, software monitoring, software instrumentation

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 a rise of software-as-a-service paradigm as opposed to traditional shrink-wrap software. These changes have led to an inherently more dynamic and fluid approach to software distribution, where users benefit from bug fixes and security updates

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...\$5.00.

instantly and without hassle. As our paper shows, this fluidity also creates opportunities for software monitoring. Indeed, additional monitoring code can be seamlessly injected into the running software without the users 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 paper 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. 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.

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. 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 run-time analyses across many users' execution of an application, such that no single user experiences

the overhead of heavy-weight 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 have 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, 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 extend to any form of client-executable code, such as Flash or Silverlight content. A public release of our prototype proxy, extensible via plug-in instrumentation policies, is available at <http://research.microsoft.com/projects/ajaxview/>.

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 paper 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.
- 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 new 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 wide 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.

1.2 Paper Organization

The rest of the paper 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 micro-benchmarks of our prototype’s performance in

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

Figure 1: Performance of simple JavaScript operations varies across commonly used browsers. Time is shown in ms to execute 10k operations.

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. We discuss implications for web application development and operations in Section 8. Finally, Sections 9 and 10 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.

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 below, this lack of visibility is exacerbated by uncontrolled client-side and third-party environment dependencies and their heterogeneity and dynamics.

Non-standard Execution Environments: While the core JavaScript language is standardized as ECMAScript [13], 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. *E.g.*, 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 Figure 1 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 [25]; and, more recently, the popular social bookmark website, `del.icio.us`, moved the URLs pointing to some of their public data streams, breaking dependent applications [7].

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 Figure 6). 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.

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 that did not exist in previous systems.

Instant redeployment: In contrast to traditional desktop software, changes can be made instantaneously to web 2.0 applications. AjaxScope takes advantage of this ability 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 [17, 20]. 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 [8]. 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.

Runtime analysis and debugging: AjaxScope provides an excellent platform for runtime analysis, from finding simple bugs like infinite loops (Section 3.2.2) to complex pro-active debugging policies such as memory leak detection (Section 6.2). Given the large number of users for the more 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 [3]. While usability evaluation is not a focus of this paper, we discuss some of the privacy and security implications in Section 8.

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

Figure 2: Policies described in above and in Sections 5–7

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 3 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 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.¹

3.2 Example Policies

Below, 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.²

¹One 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.

²Readers 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.

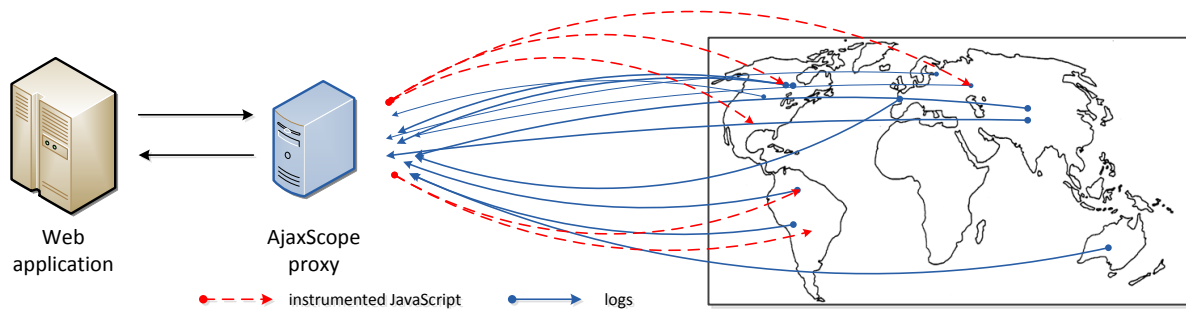


Figure 3: Deployment of AjaxScope server-side proxy for a popular web application lets developers monitor real-life client-side workloads

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 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 code below shows one common pattern leading to infinite loops [33].

```
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, alreadySend = false;
for (var i = 0; i < document.images.length; i++) {
  if (!alreadySend &&
    (++loopCount > LOOP_THRESHOLD)) {
    ajaxScopeSend('Unexpectedly long loop '
      + ' iteration detected');
    alreadySend = 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[24].

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, dependent 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 [16, 1]. Our own micro-benchmarks, shown in Figure 1, indicate that using `Array.join` instead of the default string concatenation operator `+` can produce over 130x performance improvement on some browsers.

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

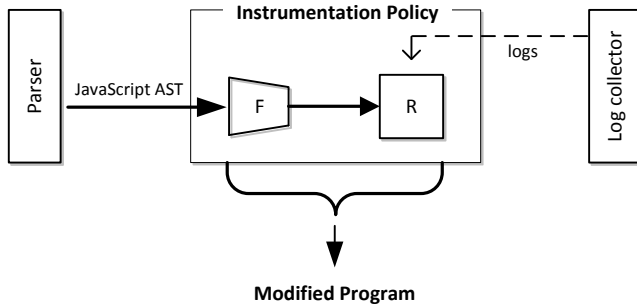


Figure 4: Structure of an instrumentation policy. The first F stage in these policies is a simple static analysis or filter to identify relevant instrumentation points. The second R stage is a rewriting node to inject instrumentation code into the program.

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 .³ Depth maintenance is accomplished by having a global hash map of `string` \rightarrow `depth` values. Since strings are passed by value in JavaScript, this approach can occasionally result in false positives, although we have not seen that in practice. 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

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

³For this rewriting, function `adjustDepth` is injected by AjaxScope into the header of every translated page.

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

Figure 5: Overhead of message logging across browsers. All times are reported in ms.

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 4.

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 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 described in this paper. 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 prototype uses a custom JavaScript parser based on the ECMA language specification [13] in C#, whose performance is further described in below.

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

Figure 6: Benchmark application statistics for Internet Explorer 7.0 (IE) and FireFox 1.5 (FF).

4.1 Micro-benchmarks

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

4.1.1 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.

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 Figure 5, our measurements show that the overhead of logging a single message is approximately 0.01–0.02 ms.

4.1.2 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 ms to parse a 10,000-line JavaScript file. Dedicated server-side deployments of AjaxScope can improve performance with cached AST representations of JavaScript pages.

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 Inter-

net. 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 application and sites from several categories, including portals, news sites, games, etc. Summary information about our benchmarks is shown in Figure 6. 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 described above, 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. Section 5 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 in AjaxScope. We then show how such adaptive instrumentation can be used to reduce the performance and network overhead of function-level performance profiling, via *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 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, *e.g.*, 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 7, 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

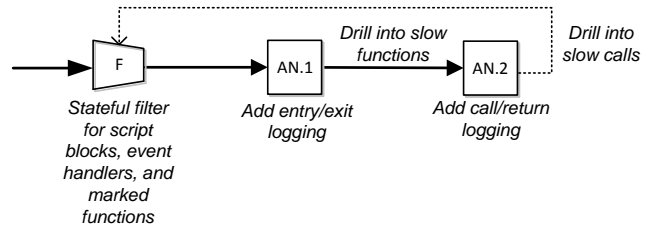


Figure 7: Policy for drill-down performance profiling

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 non-parametric 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 ms. Eventually, the drill-down process stabilizes, having instrumented all the slow functions, without having ever added 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 ms. 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 8 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 9 and 10 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). On average, drill-down adaptation alone provides a 20% (mean) or 30% (median) reduction in execution overhead. As seen in Figure 9, 7 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 lit-

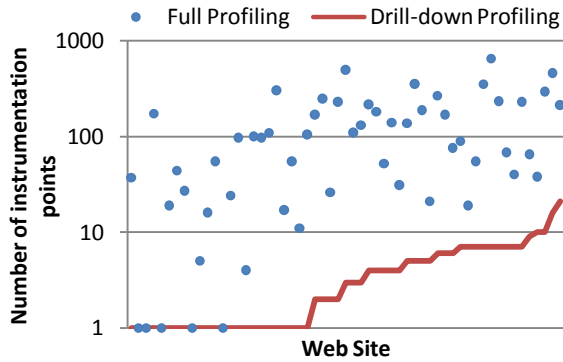


Figure 8: Number of functions instrumented per web site with full profiling vs. drill-down profiling

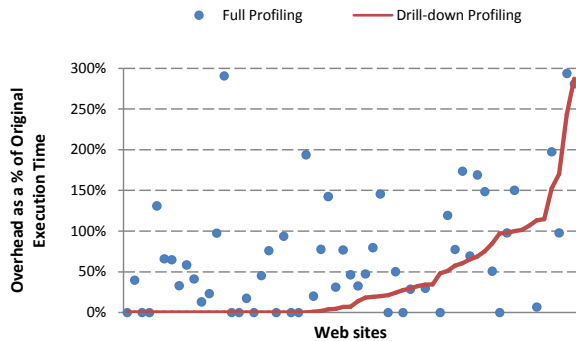


Figure 9: Execution time overhead of drill-down performance profiling compared to full performance profiling

JavaScript executing, and the measured difference in overhead is within the approximate precision of the JavaScript timestamp (around 10–20ms). 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, drops from 300KB to 64KB, and the median overhead drops 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 tests*. The purpose of a distributed test is to test for the existence or nonexistence of some condition, while spreading out the overhead of instrumentation code across many users’ executions of a web application. Note that all

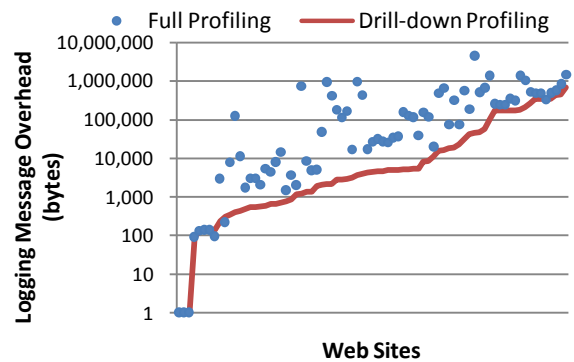


Figure 10: Logging overhead of drill-down performance profiling compared to full performance profiling

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 [15]. 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


```

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

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

```

Figure 11: An object cycle involving JavaScript and DOM objects

```

<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>

```

Figure 12: A memory leak caused by erroneous use of closures

objects, making it impossible for the garbage collector to reclaim them [29]. Other memory leaks are caused by browser implementation bugs [28, 5].

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.

Example 3. An example of such a memory leak is shown in Figure 11. DOM element whose DOM id is leaked has a pointer to global JavaScript object `global` through property `expandoProperty`. Conversely, `global` 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 11 are not the most common source of leaks in real applications, though. JavaScript closures inadvertently create these leaking cycles as well.

Example 4. Figure 12 gives a typical example of closure misuse, leading to the creation of cyclical heap structures. DOM element

referred to by `obj` points to the closure through the `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 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.

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 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.

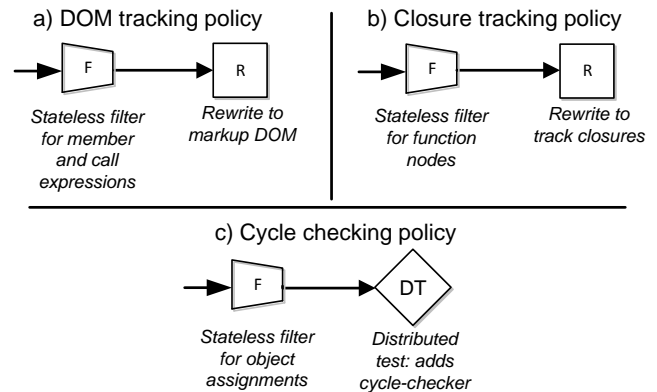


Figure 13: Three instrumentation policy pipelines work together to catch circular references between DOM and JavaScript objects that are potential memory leaks

```

352 var pipelineContainers = document.getElementById("cnnPipelineModule").getElementsByTagName("div");
...
355 for (var i=0; i<pipelineContainers.length; i++){
356     var pipelineContainer = pipelineContainers[i];
357     if(pipelineContainer.id.substr(0,9) == "plineCntr") {
358         pipelineContainer.onmouseover = function () {CNN_changeBackground(this,1); return false;}
359         pipelineContainer.onmouseout = function () {CNN_changeBackground(this,0); return false;}
360     }
... }

```

Figure 14: 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.

6.3.2 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 12 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

```

function(evt){...} → function(evt){...}.locals →
    obj → 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

```

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

```

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 cycles includes a DOM object, as determined by checking the `isDOM` property, which is set as described above.

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 .

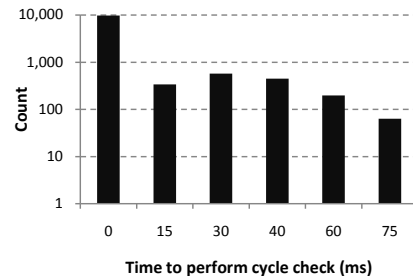


Figure 15: The histogram of circular reference check times. The vast majority of checks for cycles take under 1 ms

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 14 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 `onmouseout` 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 15 shows a histogram of the performance overhead of an individual cycle check. We see that almost all cycle checks have a minimal performance impact, with a measured overhead of 0ms. A few cycle checks do last longer, in some cases up to 75ms. 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 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 16 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 1.8sec. At 0% instrumentation probability, we reduce the startup to its baseline of 230ms. This demonstrates that simple distribution of instrumentation across users can turn heavy-weight runtime analyses into practical policies with a controlled impact on user-perceived performance.

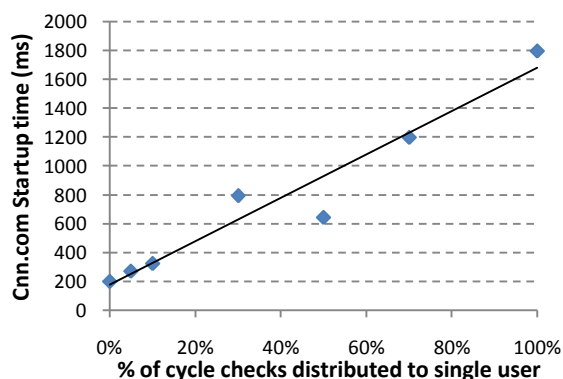


Figure 16: The average startup time for `cnn.com` increases linearly with the probability of injecting a cycle check

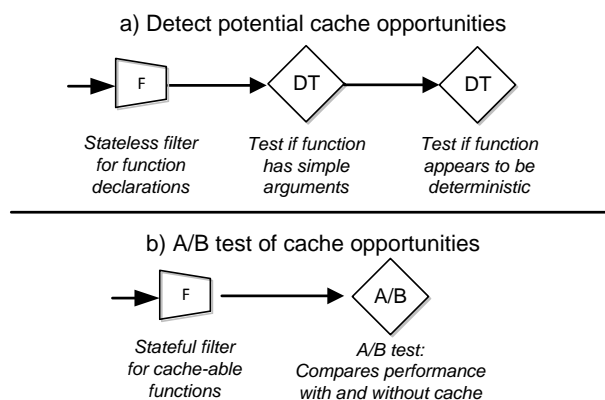


Figure 17: Two policies work together for detection and performance testing of cache opportunities. After policy (a) finds a potential cache opportunity, a human developer must check its semantic correctness before policy (b) can test it for performance improvements.

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 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 multi-stage instrumentation policy to detect possibly valid optimizations and evaluate the potential benefit of applying the optimization. Let us consider a simple optimization strategy: the insertion of function result caching. For

this optimization strategy to be correct, the function being cached must (1) return a value that is deterministic given only the function inputs and (2) have no side-effects. We monitor the dynamic behavior of the application to 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.

The first stage of such a policy injects test predicates to help identify when function caching is valid. To accomplish this, the rewriting rule essentially inserts a cache, but continues to call the original function and check its return value against any previously cached results. If any client, across all the real workload of an application, reports that a cache value did not match the function’s actual return value, we know that function is not safe for optimization and remove that code location from consideration.

After gathering many observations over a sufficient variety and number of user workloads, we provide a list of potentially 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 non-determinism. The advantage of this first stage of monitoring is that reviewing a short list of possibly valid cache-able code points should be easier than inspecting all the functions for potential cache optimization.

In the second stage of our policy, we use automatic rewriting to cache the results of functions that the developer deemed to be free of side-effects. To test the cost and benefit of each function’s caching, we distribute two versions of the application: one with the optimization and one without, where both versions have performance instrumentation added. Over time, we compare our observations of the two versions and determine when and where the optimization has benefit. For example, some might improve performance on one browser but not another. Other caches might have a benefit when network latency 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.

Figure 18 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 invalidate 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.

Function	Deterministic	Hit rate	Performance		Improvement	
			Original (ms)	Cached (ms)	ms	%
OutputEncode_EncodeURL	✓	77%	0.85	0.67	0.18	21%
DegToRad	✓	85%	0.11	0.00	0.11	100%
GetWindowHeight	✗	90%	2.20	0.00	2.20	100%
GetTaskAreaBoundingBoxOffset	✗	98%	1.70	0.00	1.70	100%
GetMapMode	✗	96%	0.88	0.00	0.88	100%

Figure 18: Results of search for potential cacheable functions in Live Maps

8. DISCUSSION

Section 8.1 presents possible deployment scenarios for AjaxScope. Section 8.2 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 8.3. Finally, Section 8.4 addresses the interaction of AjaxScope and browser caching.

8.1 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, while others may 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 paper 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.

8.2 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 *meta-monitoring*.

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.

8.3 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 a trust relationship between a user and a web application and, importantly, a strong boundary to that relationship, enforced by 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 only stored 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 [23, 31, 21], 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.

8.4 Caching Considerations

While instant redeployment enables many key AjaxScope features, unfortunately, it does not interact well with client-side caching. Indeed, many complex web applications are organized around a collection of JavaScript libraries. Once the libraries are transferred to the client and cached by the browser, subsequent page loads usually take much less time. If AjaxScope policies

provide the same instrumentation for subsequent loads, rewriting results can be easily cached.

However, since we want to perform policy adaptation or distribution, we currently disable client-side caching. The browser can check whether AjaxScope wants to provide a new version of a particular page by issuing a HEAD HTTP request. Depending on other considerations, such as the current load or information about network latency of that particular client, AjaxScope may decide whether to provide a newly instrumented version.

9. RELATED WORK

Several previous projects have worked on improved monitoring techniques for web services and other distributed systems [4, 2], 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 et al. [30], 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.

While performance profiles have been used for desktop application development for a very long time, AjaxScope is novel in that it allows developers to gain insight into application behavior in a wide-area setting. Perhaps the closest in spirit to our work is Para-Dyn [22], which uses dynamic, adaptive instrumentation to find performance bottlenecks in parallel computing applications. Much research has been done in runtime analysis for finding optimization opportunities [27, 10, 20]. In many settings, static analysis is used to remove instrumentation points, leading to a reduction in runtime overhead [20]. 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.

Both BrowserShield and CoreScript use JavaScript rewriting to enforce browser security and safety properties [24, 32]. AjaxScope's focus on non-malicious 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.

In recent years, runtime program analysis has emerged as a powerful tool for finding bugs, ranging from memory errors [26, 12, 6, 15, 10] to security vulnerabilities [14, 21, 23]. 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 [17, 18, 19]. We believe that the adaptive instrumentation of AjaxScope can improve on such algorithms by enabling the use of active learning techniques [11].

10. CONCLUSIONS

In this paper 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, in-

cluding 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 paper 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 [9] and the property of instant redeployability become more wide-spread, 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.

11. REFERENCES

- [1] String performance in Internet Explorer. <http://therealcrisp.xs4all.nl/blog/2006/12/09/string-performance-in-internet-explorer/>, December 2006.
- [2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Symposium on Operating Systems Principles*, pages 74–89, October 2003.
- [3] Richard Atterer, Monika Wnuk, and Albrecht Schmidt. 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*, pages 203–212, May 2006.
- [4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 259–272, December 2004.
- [5] David Baron. Finding leaks in Mozilla. <http://www.mozilla.org/performance/leak-brownbag.html>, November 2001.
- [6] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. *SIGPLAN Notes*, 41(6):158–168, June 2006.
- [7] Adam Bosworth. How to provide a Web API. http://www.sourcelabs.com/blogs/ajb/2006/08/how_to_provide_a_web_api.html, August 2006.
- [8] Ryan Breen. Ajax performance. <http://www.ajaxperformance.com>, 2007.
- [9] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A cache-based system management architecture. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, May 2005.
- [10] Trishul M. Chilimbi and Ran Shaham. Cache-conscious coallocation of hot data streams. *SIGPLAN Notes*, 41(6):252–262, 2006.

- [11] David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, 1996.
- [12] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Conference*, pages 63–78, January 1998.
- [13] ECMA. ECMAScript Language Specification 3rd Ed. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>, December 1999.
- [14] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*, pages 303–311, December 2005.
- [15] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 156–164, October 2004.
- [16] 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/>.
- [17] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.
- [18] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*, 32(10):831–848, 2006.
- [19] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 46–56, November 2006.
- [20] Michael Martin, Benjamin Livshits, and Monica S. Lam. 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*, October 2005.
- [21] Michael Martin, Benjamin Livshits, and Monica S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, October 2006.
- [22] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The ParaDyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.
- [23] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.
- [24] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, December 2006.
- [25] Steve Rider. Recent changes that may break your gadgets. <http://microsoftgadgets.com/forums/1438/ShowPost.aspx>, November 2005.
- [26] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebe. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 303–316, December 2004.
- [27] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Notes*, 37(1):140–153, 2002.
- [28] Isaac Z. Schlueter. Memory leaks in Microsoft Internet Explorer. <http://isaacschlueter.com/2006/10/msie-memory-leaks/>, October 2006.
- [29] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Estimating the impact of heap liveness information on space consumption in Java. In *Proceedings of the the International Symposium on Memory Management*, pages 64–75, June 2002.
- [30] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Automatic on-line failure diagnosis at the end-user site. In *Proceedings of the Workshop on Hot Topics in System Dependability*, November 2006.
- [31] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl*. O’Reilly and Associates, Sebastopol, CA, 1996.
- [32] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 237–249, January 2007.
- [33] Nicholas C. Zakas, Jeremy McPeak, and Joe Fawcett. *Professional Ajax*. Wrox, 2006.