4-24-2014

Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?

Fahad A. Arshad

Amiya K. Maji

Sidarth Mudgal

Saurabh Bagchi

Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?

Fahad A. Arshad

Amiya K. Maji

Sidharth Mudgal

Saurabh Bagchi

TR-ECE-14-05

April 24, 2014

Purdue University

School of Electrical and Computer Engineering

465 Northwestern Avenue

West Lafayette, IN 47907-1285

Is Your Web Server Suffering from Undue Stress due to Duplicate Requests?

Fahad A. Arshad, Amiya K. Maji, Sidharth Mudgal, Saurabh Bagchi,
Purdue University
{*faarshad, amaji, smudgals, sbagchi*}*@purdue.edu*

Abstract

An important, if not very well known, problem that afflicts many web servers is duplicate client browser requests due to server-side problems. A legitimate request is followed by a redundant request, thus increasing the load on the server and corrupting state at the server end (such as, the hit count for the page) and at the client end (such as, state maintained through a cookie). This problem has been reported in many developer blogs and has been found to afflict even popular web sites, such as CNN and YouTube. However, to date, there has not been a scientific, technical solution to this problem that is browser vendor neutral. In this paper, we provide such a solution which we call GRIFFIN. We identify that the two root causes of the problem are missing resource at the server end or duplicated Javascripts embedded in the page. We have the insight that dynamic tracing of the function call sequence creates a signature that can be used to differentiate between legitimate and duplicate requests. For efficiency reasons, instead of raw function call sequence, we use the function call depth as the signal and apply an efficient autocorrelation computation to detect the duplication. We apply our technique to find unreported problems in a large production scientific collaboration web service called HUBzero, which are fixed upon reporting the problems. Our experiments show an average overhead of 1.29X for tracing the PHP-runtime on HUBzero across 60 unique HTTP transactions. GRIFFIN has zero false-positives (when run across HTTP transaction of size one and two) and an average detection accuracy of 78% across 60 HTTP transactions.

1 Introduction

We live in a world where web page views are worth bragging rights and cold hard cash. Big web sites tout the number of page views. Also, provisioning of servers for hosting web content is done by monitoring the number of page requests. If the number trends high, a decision is made to provision more servers. This is typically done through human deliberation, but in some leading edge deployments, through automatic means as well [8, 5]. Web page views are monetized through various means, such as, increasing the amount charged to an advertiser for ad placement on the page, increasing the number of advertisements shown for the page, and so on. Web page views are calculated, simply put, by tracking the number of requests sent by client web browsers for that page. Could this simple but fundamental view of the web world be afflicted by a little known problem?

The affliction of duplicated web requests Yes indeed! *The affliction is duplicate web requests.* In this, the client web browser sends two requests for the same web page, the second being a redundant duplicate request. This affliction does not affect poorly run web sites alone. It afflicts two of the top 10 most visited sites — CNN and YouTube [27]. Our tests (with Chrome) show that at least 22 out of top 98 (on April 4, 2014) globally ranked Alexa [1] web sites give a duplicate request on accessing their homepages. On the academic side, we found that it affects HUBzero, a widely used open source software platform (originating from Purdue) for building powerful Web sites that support scientific discovery, learning, and collaboration [24]. The duplicated request issue causes two obvious problems. *First*, there is a spike in the traffic directed to the web server, caused by fruitless requests. For a web site that receives lots of views, this doubling can have a crippling effect due to increasing the network as well as the computational load. The increase in computational load becomes significant due to the fact that many content-rich pages today are dynamically generated by running complex, demanding scripts at the server end. *Second*, there is the potential problem of user state corruption. If the web site is tracking state, either by cookies or in another way, there is the possibility of corrupting this data.

Why do duplicate web requests happen? There are two root causes for the problem of duplicate web requests, which have been separately pointed out in many

developer forums and blog posts [3, 4, 29]. The first cause is the incorrect way in which browsers handle missing component names, or empty tags, such as, `<img src="">`, `<script src="">`, and `<link href="">`. Equivalently, this could be caused by JavaScript which dynamically sets the `src` property on either a newly created image or an existing one:

```
1 var img = new Image();
2 img.src = ""; // More realistically, the RHS will
      be some code that will resolve to the empty
      string
```

The most readable and comprehensive treatment of this first cause can be found in [3]. We will refer to this first root cause as *missing resource cause*. The second cause is the same Javascript being included in the page twice, or more number of times [27]. This is the root cause behind the duplicate web requests in CNN and YouTube. Two main factors increase the odds of a script being duplicated in a single web page: team size and number of scripts. It takes a significant amount of resources to develop a web site, especially if it is a top destination. In addition to the core team building the site, other teams contribute to the HTML in the page for things such as advertising, branding, and data feeds. With so many people from different teams adding HTML to the page, it is easy to imagine how the same script could be added twice, e.g., CNN and YouTube's main pages have 11 and 7 scripts respectively. A plausible scenario is two developers are contributing JavaScript code that requires manipulating cookies, so each of them includes the companys *cookies.js* script. Both developers are unaware that the other has already added the script to the page. This increases the time for the page to load along with the duplicate web requests problem. We will refer to this second root cause as *duplicate script cause*.

How to fix the problem? The "missing resource cause" happens because the HTML specification, version 4 [6][1] is silent on this seemingly esoteric aspect. Even though the specification indicates that the `src` attribute should contain a Uniform Resource Identifier (URI), it fails to define the behavior when src does not contain a URI. Consequently, different browsers behave in different ways. For example, Internet Explorer (IE) sends the duplicate request to the directory of the page rather than the page itself, while Firefox and Chrome send the duplicate request to the page itself. Further, the behavior of different browsers for handling different missing resources is different, *e.g.*, IE does not initiate a duplicate request with missing `script` while Firefox and Chrome do. The overall approach to handling this could be to write server-side code that will catch a similar request arising close in time to the original request and correlated with finding a missing URI in a tag. However, due to the differences in browser behaviors and for different tags, this would lead to ungainly code, with case statements for a large number of different cases. An indirect evidence comes from the fact that though this problem has been known for a while (since at least 2009), this solution is seldom deployed. The "duplicate script cause" of course has no easy solution available currently. The solution is mainly process-based — enabling better communication and coordination between developers writing or using scripts to create web pages. Thus, hopefully, the situation where two different developers include the same script on the same page or, more subtly, incorporate different but overlapping scripts on the same page, can be avoided.

Our solution approach In this paper, we present a general-purpose solution to the above problem, in a system called GRIFFIN[2]. By "general-purpose", we mean that the solution applies unmodified to all kinds of resources and browsers. The solution has at its heart the observation that the duplicate web requests cause a repeated signal, for some definition of "signal". The signal should be defined such that it can be easily traced in a production web server, without impacting computation or storage resources and without needing specialized code insertion. We find that the *function call depth* is the signal that satisfies these conditions, while preserving enough fidelity that the repeated sequence can be easily and automatically discerned. To automatically discern the repeated pattern, we use the simple-to-calculate autocorrelation function for the signal and at a lag, equal to the size of the web request (in terms of number of HTTP commands), GRIFFIN sees a spike in autocorrelation which it uses to flag the detection.

When tested over a wide range of buggy and non-buggy behavior, we find that GRIFFIN performs well with respect to both the detection and the false positive. For example, we evaluate GRIFFIN on the production NEEShub web portal at Purdue, which is the portal created out of an ongoing NSF center called NEES, for providing computational tools and data upload/download facilities to earthquake scientists and engineers all over the US [19]. NEEShub has been operational since 2009 and has had 105,000 users over the last 12 month period. So a problem in it cannot be easily dismissed as a corner case in an obscure site. We find that GRIFFIN has no false positive and an 78% detection accuracy. To make GRIFFIN feasible in real production settings, we adopt a mix of synchronous and asynchronous approaches, both without modifying the application's source code, or even

[1] HTML4 is the latest version of the specification, except for a W3C "Candidate Recommendation" for HTML5 dated 04 February, 2014.

[2] GRIFFIN is a mythical creature with the front legs, wings, and head of a giant eagle, and the body, hind legs, and tail of a lion. It is often used to guard treasures.

needing access to the source code. Synchronously we capture the call stack depth, using a built-in functionality, in the tracing tool called SYSTEMTAP. The SYSTEMTAP tool is highly efficient and has already been used in prior efforts for analyzing the properties and behavior of software systems [14]. Then, asynchronously, GRIFFIN calculates the autocorrelation function for various lags, filters the values, and flags a detection when the value exceeds a threshold. In addition to detection, GRIFFIN also provides some diagnostic insight, *i.e.*, gives an idea of the module where the root cause lies. It does this by inserting probes through SYSTEMTAP, determining the lag at which the autocorrlation function has a peak, and correlating the two to determine the suspect module.

Our contributions in this paper can be summarized as follows.

1. We provide automatic detection of duplicate web requests in a manner that is generic to any web server and works across different web clients and different root causes of the problem.

2. We develop code to extract the signal from amidst a plethora of tracing data. We zoom in on the right signal to use through insights born out of troubleshooting web servers. Our method has in fact been merged within the SYSTEMTAP code repository.

3. We evaluate our scheme with a popular production web portal for science. We report on the performance overhead as well as error coverage from our evaluation.

2 Example Bug Case

The manifestation of the duplicate web request can be silent or non-silent. Silent implies there is no visual indication of the problem at the client web browser, while in the non-silent case, there is such an indication. With the NEEShub home page, we observed a non-silent manifestation whereby multiple images are not shown as depicted in Figure 1. An example of the silent case is that the browser after downloading the multiple Javascripts, generates duplicate web request from the multiple Javascripts.

Here we present a bug-case that was observed for the beta release of the main web portal of our NSF center called NEEScomm, meant for providing a cyberinfrastructure for earthquake engineers and scientists throughout the US `www.nees.org`. GRIFFIN was able to detect it before the code update made it to the production site, and thus avoided the duplicate request problem. On accessing the homepage, the images that appear as part of background were missing (Figure 1). Listing 1 presents the code modifications that fixed the problem

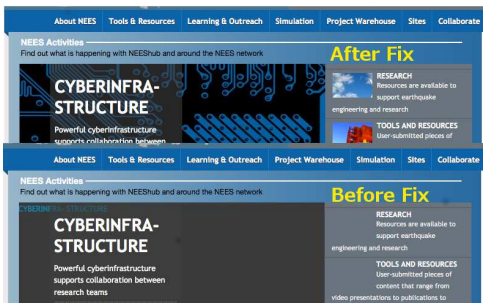


Figure 1: Duplicate bug-manifestation (with missing images) before and after the fix

(no duplicate requests seen from client). In Listing 1, `$slide->mainImage` variable does not resolves to the image `XYZ.jpg` location. Instead, it resolves to the NUL character. Manual inspection revealed that the images were missing. To verify, we hard-coded a valid image location and it fixed the duplicate problem. Listing 2 shows the runtime state of the rendered HTML in Firefox browser. On lines 3 and 10, the empty `url()` is observed, while on line 4, the `src` field in `<img>` tag having a value of "/" pinpoints the root cause for the duplicate request to the base URL.

```
1  --- a/modules/mod_fpss/tmpl/Movies/default.php
2  +++ b/modules/mod_fpss/tmpl/Movies/default.php
3  - <span style="background:url(<?php echo $slide->mainImage; ?>) no-
       repeat;">
4  + <span style="background:url(media/system/images/XYZ.jpg) no-repeat;">
5  - <img src="<?php echo $slide->mainImage; ?>" alt="<?php echo $slide->
       altTitle; ?>" />
6  + <img src="media/system/images/XYZ.jpg" alt="<?php echo $slide->
       altTitle; ?>" />
7  - <span class="navigation-thumbnail" style="background:url(<?php echo
       $slide->thumbnailImage; ?>) no-repeat;"> </span>
8  + <span class="navigation-thumbnail" style="background:url(media/system/
       images/XYZ.jpg) no-repeat;"> </span>
```

Listing 1: Code modification to fix unnecessary duplicate requests

```
1  <div class="slide" style="position: absolute; opacity: 0; z-index: 89;"
       >
2    <a class="slide-link" href="/fpss/track/35/L3Jlc291,,">
3      <span style="background:url() no-repeat;">
4        <img alt="NEEShub Release 5.0" src="/">
5      </span>
6    </a>
7  .
8  .
9  .
10 <span class="navigation-thumbnail" style="background:url() no-repeat;">
       </span>
```

Listing 2: Runtime state of generated HTML as observed by Firebug

To understand how current browser versions (Chrome 32, Firefox 26) behave under unexpected input, we did a synthetic injection in `HTML` tags: `<span style:background=X>`, `<img src=X>`, `<script src=X>`, `<iframe src=X>`, `<link href=X>`. Here `X`, the injected character, had ASCII codes in the range 32-126 excluding alphanumeric characters. We found that, in addition to duplicate requests due to empty strings which have been reported before [3], the characters '?' and '#' also resulted in duplicate requests. `<span style:background=SPACE,EMPTY>` resulted

in a duplicate request for both browsers. For Firefox, `<img src=SPACE>`, `<script src=SPACE,EMPTY>`, and `<link href=SPACE>` created duplicate requests. These injections provide evidence that browsers do behave differently and erroneously under unexpected special characters for URIs.

3 Design

Here we detail the design of GRIFFIN to detect duplicate web requests. At a high level, it comprises three steps: model application behavior at the web server (in terms of the function calls and returns), create a signal of the function call depths, and compute the auto-correlation of the signal to trigger detection. Figure 2 shows these steps in GRIFFIN.

3.1 Synchronous Tracing

Tracing can be divided into static or dynamic tracing. Static tracing involves modifying the application's source code to insert tracing statements followed by compilation and execution. Dynamic tracing involves instrumentation of live, in-production applications without needing to stop and restart them. Dynamic tracing can be sub-divided into asynchronous or synchronous tracing. Asynchronous tracing takes samples at regular intervals, from a running application, easing the possible impact on performance but also opening up the possibility of missing important events between samples. Synchronous tracing captures pre-defined events (function calls in our case) within the source code. In this work, we use synchronous tracing as it meets our following tracing framework goals.

1. The tracing should be done dynamically, i.e., the tracer should be able to connect and disconnect to an already running application without the need of stopping, recompiling and restarting.

2. The application should not be polluted with instrumentation within its source code.

3. The instrumentation should provide enough function call context for triggering post-detection diagnosis, e.g, the name of the called function, filename, classname of the object calling the function etc.

4. The instrumentation overhead should be small enough to debug problems in the production environment in an online setting.

We leverage SYSTEMTAP [17], a tracing/probing framework that can provide synchronous tracing data on Linux hosts. SYSTEMTAP is built on the same architecture as the well-known DTrace [12] tool for Solaris systems and can provide event-tracing across the whole system stack: kernel, applications, system-services, interpreters (PHP, Python, Perl, Java), databases, etc. This ability to look through the whole system with low probe-point programming overhead makes SYSTEMTAP a better fit than other tools like PIN [23] and Valgrind [26].

```
1
2  probe process("/usr/lib/apache2/modules/libphp5.so").provider("php").
       mark("function__entry")
3  {
4          printf("PHP: %d %d => %d %s file:%s line:%d classname:%s\n",
               gettimeofday_us(), tid(), thread_indent_depth(1),
               user_string_quoted($arg1), user_string_quoted($arg2),
               $arg3, user_string_quoted($arg4));
5  }
6
7  probe process("/usr/lib/apache2/modules/libphp5.so").provider("php").
       mark("function__return")
8  {
9          printf("PHP: %d %d <= %d %s file:%s line:%d classname:%s\n",
               gettimeofday_us(), tid(), thread_indent_depth(-1),
               user_string_quoted($arg1), user_string_quoted($arg2),
               $arg3, user_string_quoted($arg4));
10
11 }
```

Listing 4: `php.stp` SYSTEMTAP script with function-entry/return probes

```
1
2  PHP: 1392668507729050 22061 => 1 "a" file:"/www/a_b_c.php" line:18
       classname:""
3  PHP: 1392668507729120 22061 => 2 "b" file:"/www/a_b_c.php" line:5
       classname:""
4  PHP: 1392668507729134 22061 => 3 "c" file:"/www/a_b_c.php" line:11
       classname:""
5  PHP: 1392668507729146 22061 <= 2 "c" file:"/www/a_b_c.php" line:11
       classname:""
6  PHP: 1392668507729158 22061 <= 1 "b" file:"/www/a_b_c.php" line:5
       classname:""
7  PHP: 1392668507729167 22061 <= 0 "a" file:"/www/a_b_c.php" line:18
       classname:""
```

Listing 5: Output of `php.stp` SYSTEMTAP script

To enable tracing, SYSTEMTAP allows to write probe-point scripts. Probe-point scripts tell SYSTEMTAP two things. *(1). What event do you want to trace? (2). What do you want to print at the traced event-location?*. An example output of tracing function calls in PHP for the program in Listing 3 with SYSTEMTAP tracer-script in Listing 4 is shown in Listing 5. `abc.php` invokes the function call-chain (a()→b()→c()) from main-method. `php.stp`, that traces `abc.php`, has two probe-points, for function-entry and function-return. At both entry and return, `php.stp` logs, in order of their appearance in the `printf` call, timestamp, thread-id, function call depth, funcation name, file name, line number, and class name, if available. Other than `thread_indent_depth(long)` function, all the other functions are natively available in SYSTEMTAP .

```
function a() {
 echo "Func a";
 b(); } a();
```

```
function b() {
 echo "Func b";
 c(); }
```

```
function c() {
 echo "Func c";
      }
```

Listing 3: `abc.php`, where a() calls b() and b() calls c()

3.2 Modeling Application Behavior

For modeling purposes, we define a numeric metric called *function call-depth* that represents the runtime

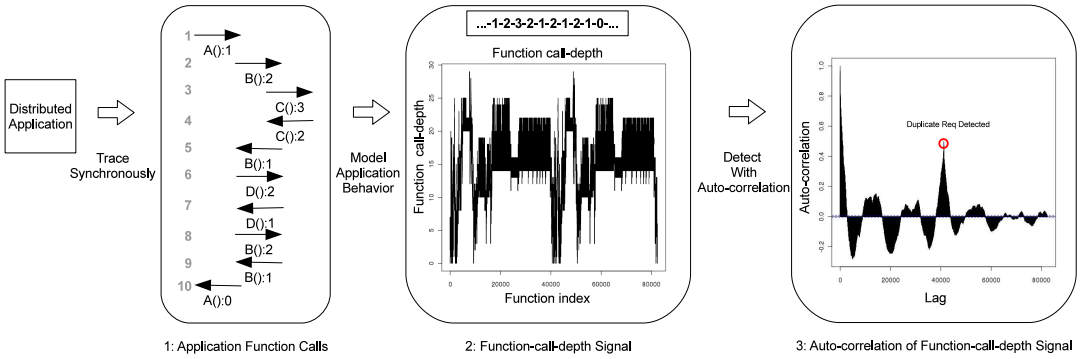


Figure 2: Overview of the duplicate-detection workflow.

function call-depth. At every function-call, the call-depth is incremented and at every return, it is decremented. *Our foundational intuition for modeling application behavior is that the flow of an application can be roughly represented by how function call depth changes.* The function call depth sequence for a given high-level web operation can be considered as a fingerprint of the high-level operation. For further exploration of this intuition, let us first define some terms: *web-request, web-click, http-transaction*. Starting from the lowest level, a web request is the HTTP request sent by the web browser, such as, GET and POST. A web click is a human user clicking in the browser to send web requests. A single web click can generate multiple web requests. A set of web clicks done in a particular sequence, as permitted by the workflow in the website, is called an http transaction. An http transaction can consist of one or more web clicks; in typical usage this will be more than one web click. An example of an http-transaction of size two is going to the homepage followed by going to the login page (HomePage→Login).

Now coming back to our intuition for detecting duplicate web requests, consider that a duplicate web request will create a duplicated signal of the function call depths. It is easy to concoct a synthetic example where this intuition is violated. For example, consider two legitimate consecutive web clicks and the corresponding web requests: (a (b (c c') b') a') (d (e (f f') e') d') giving a call-depth sequences of (1 2 3 3 2 1) (1 2 3 3 2 1). This would give the appearance to GRIFFIN of duplicated web requests. However, we find that for real web pages, the length of web clicks in terms of the number of function calls and returns tends to be much larger. This kind of accidental matching of the function call depth signal happens only very rarely for these real situations.

To get the call-depth at runtime, we add a function called `thread_indent_depth(long)` to SYSTEMTAP 's native scripts. This function returns a number corresponding to the depth of nesting. We call this function `thread_indent_depth(1)` in the probe-point SYSTEMTAP script (Listing 4). Here, the argument one means that at every function-call, increment the depth by one. We submitted this function to the SYSTEMTAP repository and it has been merged (commit-id:cecdb2dddae55b510814dc8a6d7510e5fa5e0d9f) into SYSTEMTAP 's master-branch and is available out-of-the-box after SYSTEMTAP is installed [9].

3.3 Duplicate Detection Algorithm

With the function call-depth sequence captured, the next goal is to detect whether the sequence has a repetitive pattern and to do this efficiently with respect to time. To do this, we use a common signal analysis technique to detect repeating patterns, *auto-correlation* [31] of the function call-depth signal. Auto-correlation of a signal x is defined by R_{xx} (Equation 1) as a function of lag-value t, where t varies from zero (perfect signal match with R_{xx}=1) to n, the sequence length in terms of the number of function calls and exits. Ideally for GRIFFIN to detect duplicate web requests resulting from a single user web click, it would be possible to segment the web requests for each web click. But that is not always possible in practice, as we discuss in Section 4.1. Auto-correlation can be viewed as a sequences of *shift, multiply, sum* operations for all lag values on function call-depth signal. Intuitively, we are using auto-correlation to estimate the similarity between the signal and its time shifted versions for various values of the time shift. If the function-depth signal is exactly repeated twice, we expect to see a peak of 0.5 around the lag value of *n/2*.

$$R_{xx}[t] = \frac{C_t}{C_0} \text{ where t=0,...,n}$$
$$C_t = \frac{1}{n}\sum_{s=max(1,-t)}^{min(n-t,n)} [X_{s+t} - \bar{X}][X_s - \bar{X}] \quad (1)$$

We present the auto-correlation-based duplicate-detection algorithm in Figure 3. After auto-correlation computation for all lag-values, we find the index at which

```
/* Compute auto-correlation for all lags */
X ← load signal values
X̄ ← get mean value
C_0 ← 0
R_xx[t] ← get an empty array of size n
Threshold ← 0.4
for each t in range n:
  sum ← 0
  for j ← 0; j < n; j++:
    sum ← sum + (X[j]−X̄)∗(X[j+t]−X̄)
  sum ← sum/n
  if t = 0
    C_0 ← sum
  R_xx[t] ← sum/C_0

/* Get the index where auto-correlation first
     becomes negative */
index ← 0
for each t in range n:
  if R_xx[t] < 0
    index ← t
    break

/* Check if auto-correlation is great than
     threshold beyond index */
for i ← index; to end of sequence
  if R_xx[t] ≥ Threshold
    Print Duplicate Request Detected!
    break
```

Figure 3: Algorithm to detect duplicate messages from function call-depth signal.

```
global remote_ip

//Probe 1: Apache probe to get ip-address of incoming web-request
probe process("/usr/*bin/apache2").function("ap_process_request")
{
        remote_ip[tid()]=user_string(@cast($r->connection, "conn_rec")
              ->remote_ip)
}

//Probe 2: probe that receives the http-request from Apache in the PHP
      runtime
probe process("/usr/lib/apache2/modules/libphp5.so").function("
      php_apache_request_ctor")
{
        printf("APACHE: %d %d %s\n", gettimeofday_us(), tid(),
              user_string_quoted($r->unparsed_uri));
}

//Probe 3: php function-entry
probe process("/usr/lib/apache2/modules/libphp5.so").provider("php").
      mark("function__entry")
{
        printf("PHP: %d %d %s => %d %s file:%s line:%d classname:%s\n",
              gettimeofday_us(), tid(), remote_ip[tid()],
              thread_indent_depth(1),user_string_quoted($arg1),
              user_string_quoted($arg2), $arg3, user_string_quoted(
              $arg4));
}

//Probe 4:
probe process("/usr/lib/apache2/modules/libphp5.so").provider("php").
      mark("function__return")
{
        printf("PHP: %d %d %s <= %d %s file:%s line:%d classname:%s\n",
              gettimeofday_us(), tid(), remote_ip[tid()],
              thread_indent_depth(-1),user_string_quoted($arg1),
              user_string_quoted($arg2), $arg3, user_string_quoted(
              $arg4));
}
```

Listing 6: SystemTap probes for tracing

the auto-correlation first becomes negative, call this t_0. For values of auto-correlation beyond t_0, we find if there is any value greater than a threshold value τ. If yes, we flag a duplicate-detection. For the duplication of a set of web requests once, we expect ideally an auto correlation peak of 0.5. But to tolerate the normal variation in function call-depth signal, we set the threshold τ to be a little lower than 0.5. We report on our sensitivity empirical study in Section 5.3. The reason for starting the search beyond t_0 is that then we eliminate the high values of autocorrelation that we will see due to the original signal being correlated with itself with small time lags.

3.4 Usage Modes

We envision GRIFFIN to work in two scenarios, pre-production *testing* and *in-production*. In testing, developer's have control of the environment and trace segmentation is not an issue. Here, a possible concern by developers could be GRIFFIN's detection latency, which is in order of seconds. Also, there are several works that show speed-up of autocorrelation-like functions using parallelization in software [11] and hardware like FPGAs [21], GPUs [22]. For in-production mode, operators' main concern could be the overhead of configuring and tuning GRIFFIN and the application tracing overhead, which is incurred in the critical path of all web requests and responses. GRIFFIN's configuration is minimal with only one threshold parameter for which we provide a recommendation (threshold=0.4) with our sensitivity analysis. In fact, this simplicity was appealing enough for us that we adopted this scheme in favor of more complex, and potentially better-performing, schemes that have a plethora of parameters. The tracing overhead with SYSTEMTAP is low enough as it is. To further minimize the tracing overhead, an operator can run GRIFFIN in time intervals of low load on the web server .

4 Experimental Setup

4.1 Configurations: Hardware, Software, Tracing

NEEShub infrastructure is running Apache/2.2.16 (Debian) web server in Prefork MPM (Multi-Processing Module) [2] mode, *i.e.*, with multiple processes and one thread per process, on a VM with Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz with 6GB RAM. The PHP-runtime (`libphp5.so`) version is 5.3.3 and is compiled with `--enable-dtrace` option in order for SYSTEMTAP (ver 2.4) to be able to intercept PHP-function calls and returns with its probes.

Listing 6 presents the SYSTEMTAP probes used for synchronous tracing. The objective that we discuss here is how to segment multiple users so that the analysis can be done on web clicks from a single user. This segmentation is conceptually done using a combination of fields such as, source IP address, port, etc. For ease of exposition, in the discussion in this section we will use IP address as the proxy for this combination. In Listing 6, we show the SYSTEMTAP probes that GRIFFIN inserts. On

receiving a web request, Apache probe (Probe 1) fires first. When probe 1 fires, we record the IP address of the incoming web-request in a global variable with the key equal to the Apache thread id. Since the design is such that the same thread id is going to complete the web click request, so, whenever subsequent probes fire they have the same thread id. The probe at the interface between the Apache web server and the PHP module is probe 2. Probe 3 gets triggered at all function calls in the PHP module and probe 4 at all function returns in the PHP module. In probes 2, 3, and 4, we read the global variable to get the IP address corresponding to thread id of the current thread. Thus, even though the IP address is visible only to probe 1, inward-situated probes 2-4 can also access the client-distinctive information. In terms of frequency, most probe 1's have a corresponding probe 2; each probe 2 gives rise to many probes 3 and 4. This is due to the design, common to most content-rich web sites, that dynamic content is generated through the PHP module and the PHP module invocations traverse a long chain of libraries. For example, for the NEEShub, on an average a single web click results in a total of 67,071 function calls and returns in the PHP module.

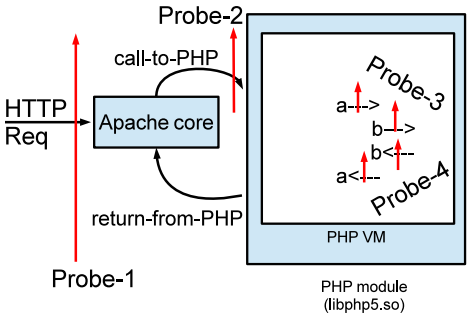


Figure 4: Probes that fire in the life-cycle of an HTTP request

4.2 Evaluation Metrics

We evaluate GRIFFIN's detection performance with traditional definitions of accuracy and precision (Equation 2). We establish the ground truth through manual verification, at client-end, by checking duplicate requests for each web-click using browser debugging tools, Firebug and Chrome-dev-tools. We measure the overhead of GRIFFIN in two areas, tracing overhead and detection overhead. Tracing-overhead is the fraction of total time, taken by SYSTEMTAP 's probes while processing a given web-click. Detection overhead or detection latency is measured in the standard way as the time elapsed for all the detection steps—getting the signal as input, computing auto-correlation, determining the trigger point, and sweeping through a series of time lag values to detect if there is a peak corresponding to the duplicate web request.

$$\begin{aligned} Accuracy &= \frac{TP+TN}{TP+TN+FP+FN} \\ Precision &= \frac{TP}{TP+FP} \end{aligned} \tag{2}$$

4.3 NEEShub Infrastructure

We apply GRIFFIN to a large-scale real setup called NEEShub [19], a system built with HUBzero [24] open source software platform. HUBzero is a proven dynamic website building technology to support scientific research and educational activities. HUBzero has over 29 documented hubs [7] operating across various scientific disciplines. In addition to building dynamic websites, HUBzero provides a modular architecture that helps to quickly build tools (for data-analysis, simulation etc) for a given scientific discipline. We can view NEEShub as an instance of HUBzero where the website and tools are developed for Earthquake Engineering discipline. NEEShub architecture is composed of a front-end webserver and a back-end database. NEEShub's website usage shows an increasing yearly trend of number of unique users who logged into the website for past four years. Additionally, after its inception in 2009, the annual webserver hits for last 3 years has been over 32 million. Both, webserver hits and increasing website users mean that any duplicated web requests can cause a performance bottleneck or may lead to unneeded hardware upgrade.

5 Evaluation

5.1 Experimental Workload

GRIFFIN's testing was conducted on a replica of the production site (www.nees.org), technically referred to as a "staging machine" where developers merge their code after doing the unit testing on their own development box. We made no modifications or synthetic error injections. Therefore, we expected to find few, if any, problems with the website.

We tested GRIFFIN's duplicate-detection performance by sending a total of 60 HTTP transactions of varying sizes. The size of a transaction is measured by the number of web clicks incorporated within the transaction. Thus, the transaction HomePage→Login has a size of two. Also, for the analysis (autocorrelation computation), the signal is considered the entire transaction. We used 20 transactions for each of the sizes 1,2,3. These 60 HTTP transactions were executed following different possible user workflows as enabled by the web portal. We tried to cover all the workflows that a typical user would follow while visiting the website. This is enabled

by our having worked as part of the NEES team for 3+ years.

Ideally, the analysis in GRIFFIN will consider the traces corresponding to a single web click from a single user. The combination of IP address, source port, etc. is meant to segment different users. Within a single user, we expect that different web clicks are handled by threads of different IDs. We empirically validated that this is *always* the case for all our transactions. This is explained by the design of the Apache web server, which has a maximum number of concurrent requests that can be served, given by the parameter `MaxClients` with a default value of 256. For Apache Prefork MPM, this translates to the total number of processes. When a request processing is completed, the process becomes idle and after an expiry time, is killed off. If the next request arrives within the expiry time, then with a probability $\frac{1}{MaxClients}$, it will be handled in a process with the same thread ID. If the next request arrives after the expiry time, then there is only an infinitesimal chance that it will be processed with the same thread ID. To account for these probabilities, plus other Apache modes (multi thread, etc.), we also evaluate GRIFFIN's performance with HTTP transactions of size greater than one. For HTTP transaction size greater than one, we are mimicking the situation where the duplicate request happened due to one web-click (*e.g.*, HomePage) but GRIFFIN analyzed two (*e.g.*, HomePage→Login) or three web clicks (*e.g.*, HomePage→Login→LoggingIn) together.

5.2 Accuracy and Precision Results

Out of the 7 duplicate request problems (among the 60 HTTP transactions),GRIFFIN was able to correctly find 4 duplicated requests i.e., *HomePage, Topics-page, SimulationWiki-page and Wiki-page*. SimulationWiki page was due to a Javascript-based duplication, while the other three were due to missing-resources. GRIFFIN missed 3 cases of duplicated requests, *warehouse, simulation and education* pages.

GRIFFIN's accuracy and precision with different HTTP transaction sizes is presented in Table 1. GRIFFIN provides an average accuracy of 80% across HTTP transactions of size one and two with no false positives. With three web clicks, GRIFFIN's performance degrades– here 0% precision is misleading in the sense that out of the 20 HTTP transactions of size three, only one (HOMEPAGE→LOGIN→LOGGINGIN) had a duplicate request which GRIFFIN did not detect. GRIFFIN falsely flagged 4 out of 20 transactions giving a false positive rate of 20% for HTTP transactions of size three. The reason why GRIFFIN did not detect HOMEPAGE web-click within HOMEPAGE→LOGIN→LOGGINGIN transaction is due to the significant difference of LOGGGINGIN function call-depth signal from the signals of HOMEPAGE and LOGIN web clicks (see the increase in function call-depth signal between index 100K to 150K in Figure 6). Here, HOME and LOGIN web clicks have an average function call-depth of 15.61 and 15.47 respectively while LOGGINGIN has an average of 32.42 making it significantly different. With HTTP transaction of size 3, GRIFFIN is performing its analysis after combining these three signals into one. Thus, the divergence in the single combined signal means that the autocorrelation values, even with one duplication, tend to be low, and stay below the threshold. In practice, the HTTP transactions of size 3 will be very rare because of the discrimination that GRIFFIN will be able to do using the thread ID.

	Accuracy	**Precision**
one-click	$90\% = \frac{18}{20}$	$100\% = \frac{3}{3}$
two-clicks	$70\% = \frac{14}{20}$	$100\% = \frac{4}{4}$
three-clicks	$75\% = \frac{15}{20}$	$0\% = \frac{0}{4}$

Table 1: Summary of Performance results

With the ideal (and practically common) case of analysis over HTTP transaction of size 1, GRIFFIN shows 90% accuracy and 100% precision. As an example, the function call-depth and autocorrelation for HOME web-transaction is presented in Figure 2. We see that the autocorrelation has a clear peak value of 0.4998 near a lag-value of 40,000 which is detected by GRIFFIN (with a threshold set at 0.4). Manual checking, both at user-end and at server-end revealed that HOME web-request ("/") is being sent twice by the user's browser. Further inspection on the server revealed that a field called `hits` in the back-end database is incremented on every HOME web-transaction. We reported this hitherto unknown problem to the web developer at NEES, and it was subsequently fixed and not pushed into the production environment. Testing GRIFFIN with HTTP transactions of size 2, we

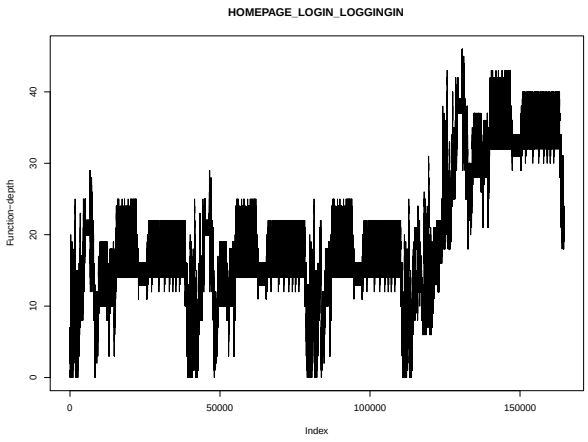


Figure 5: HOME→LOGIN→LOGGINGIN: Function call-depth signal for three web clicks from browser

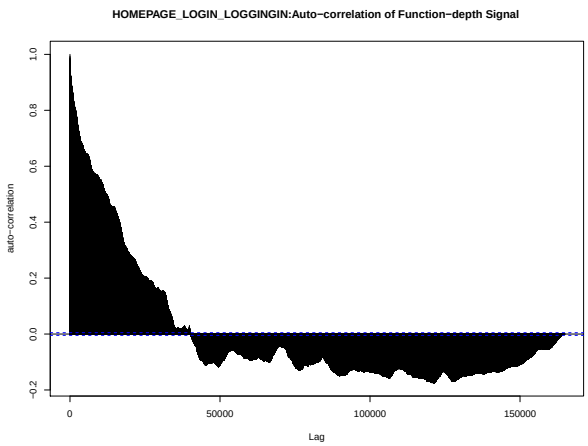


Figure 6: HOME→LOGIN→LOGGINGIN: Autocorrelation for three web clicks from browser

observe a drop in accuracy (to 70%). This happens due to the significant variability in the basic signal due to the very different nature of the function call invocations in the two web clicks. Expectedly, autocorrelating a divergent signal gives low autocorrlation values, which sometime fall below the GRIFFIN threshold, which has a default value of 0.4.

5.3 Sensitivity and Overhead

GRIFFIN's sensitivity to different parameters, sequence length, threshold and number of traced contiguous web clicks is critical from a usability perspective. With an increasing number of contiguous web clicks, GRIFFIN's accuracy and precision drop. The pattern of accuracy decreasing with increasing number web clicks holds true with increasing sizes of the traces. We present GRIFFIN's sensitivity with different thresholds in Figure 7. Looking at Figure 7a and Figure 7b, we set GRIFFIN threshold to 0.4 as the default value for GRIFFIN to provide us zero false positives, *i.e.*, 100% precision. The user can decrease the threshold for fine tuning her system, but we suggest to not go below 0.35 (based on Figure 7b) as that can result in possible false positives.

The detection latency as a function of the sequence length (*i.e.*, the number of trace events due to SYSTEMTAP probes) is shown in Figure 8. It shows the expected behavior of greater latency with increasing sequence length. This is due to a larger number of autocorrelation computations for a longer trace length. However, the upper range of the sequence length is typically about 100K and with that we have a detection latency of about half a minute, which should be fast enough to be useful for the subsequent manual process of fixing the problem. The average tracing overhead across the 60 tested HTTP transactions is 28.6% with a standard deviation of 10.0%. The overhead for HTTP transactions for each size is presented in Table 2. The tracing overhead

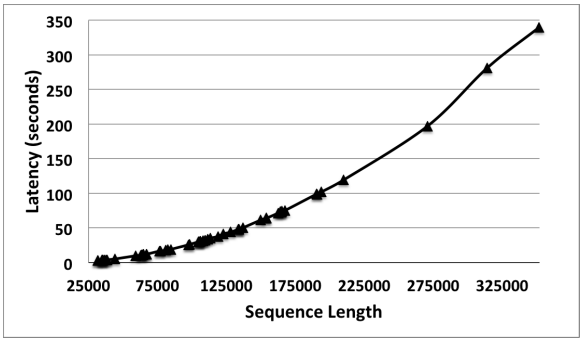


Figure 8: Detection Latency

is independent of the length of the sequence and the differences seen are due to statistical variations. This overhead can be reduced simply by removing all the fields in the traces, say as in Listing 4, except for the thread ID (needed for segmentation), function call depth (needed for detection), and filename (needed for diagnostic context, as explained below in Section 6).

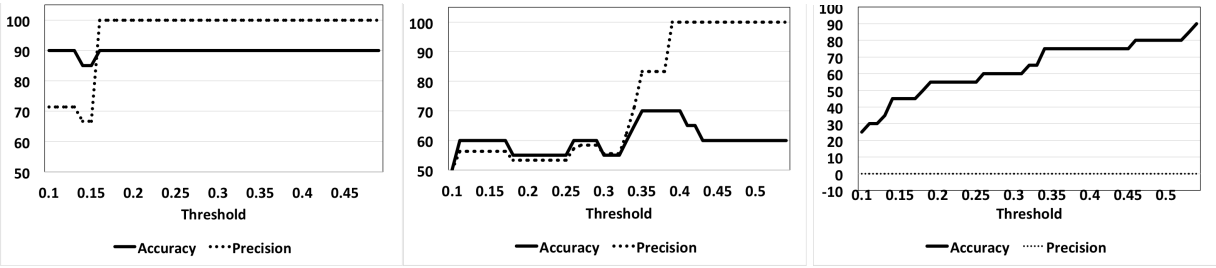
	Tracing Overhead (Avg)	Tracing Overhead (Std. Dev)	Sequence Length (Avg)	Sequence Length (Std. Dev)
one-click	24.0%	6.6%	67,071	54,165
two-clicks	32.8%	11.6%	131,511	76,630
three-clicks	29.1%	9.1%	141,427	33,727

Table 2: Tracing Overhead

6 Discussion

GRIFFIN's Diagnostic-context: When GRIFFIN detects duplicate web-requests, a diagnostic-context about the detection would help the developers as a starting point for debugging. At detection-time, in addition to the autocorrelation value, we also have the lag when this autocorrelation value exceeded the threshold, call this t_{max}. We use t_{max} alongwith the information provided by probe-2 (Figure 4), a probe that records the HTTP-request going from `apache-core` to PHP-runtime, to provide the *diagnostic-context*. With the t_{max}, we get the nearest next fired probe-2 and extract a high-level component (module name) from the file name. For the duplicate bug of Figure 1, this simple scheme is able to flag `mod_fpss` module in Joomla, the Content Management System, on which HUBzero is built.

ID-based Trace Segmentation: Given two parallel web-clicks from two different users, the segmentation-problem (as discussed in Section 5) of getting a per-user click-trace can be solved using a variety of IDs available at the server-end. Specifically, for Apache server, the data structures, `request_rec` (created whenever the server accepts an HTTP request from client) and `conn_rec` (an internal representation of TCP connections in Apache) can help in filtering. An interesting scenario occurs when the same IP address represents different users, *e.g.*, multiple users behind a NAT (Network Address Translation) server. In this case simply segment-



(a) One-click (b) Two-click (c) Three-click

Figure 7: Sensitivity of GRIFFIN for each of one, two and three-clicks

ing users by IP address will not work and additional information, *e.g.*, port number is needed. This can be achieved with writing an Apache probe with SYSTEMTAP intercepting the function `ap_process_connection` (which is invoked in the call-chain for every incoming web-request) and reading the port field using `cast($c, "conn_rec")->remote_addr->port`.

Simple Strawman Schemes: Given that we have the function call depth and the sequence length, a simple scheme for detection of duplicate requests can be a threshold for each of the two parameters. Though simple in theory, we will need to maintain and learn per-web-request type thresholds, *e.g.*, different thresholds for each of Homepage, Login etc. Additionally, as the application is developed new web-requests would have to be benchmarked for learning the threshold. We think that for a busy operator, this poses too much of configuration overhead. An attraction of GRIFFIN is that it requires little configuration for detecting a wide variety of duplicate request problems. Another possible technique is to use logs at the server and use some heuristic to flag detection when similar requests are received within a small time window. Such a scheme though would be fragile in practice and be unable to handle natural variations in user request patterns.

7 Related Work

Most of the existing approaches to handle duplicate requests are *not* at the application-level. TCP [13] is the classic example that uses sequence numbers along with a windowing-based mechanism to do duplicate detection of IP packets. Stateless protocols like HTTP have to deal with the request-response nature and maintain state at the application-level. Application-level works include similarity detection [28] deployed at web-proxy caches to eliminate redundant network traffic, duplicate-content detection [30] with clustering and similarity metrics [16]. These are directed at generic payloads and are therefore less accurate than GRIFFIN in general. Another related area is schemes for avoiding the occurrence of duplicate requests in the first place, which are complementary to GRIFFIN. These schemes are implemented either on the client-end [20] or on the server-end [25].

Finding relevant system events to detect and diagnose failures is often equated to the problem of finding a needle in a haystack. Over the last decade, several researchers have proposed solutions to this challenging problem [15, 32, 10, 18]. The high-level objective here is to mine vast amounts of system data to find relevant signatures for failures. Once "syndromes" [15] or signatures are created [32], these can be used to detect problems in the future efficiently. Present day data centers often suffer from tens of minutes to hours of downtime due to inherent difficulties in diagnosing and fixing such failures. [10] and [18] partially automate this process, thereby, reducing manual effort and downtime. Our work falls within this broad umbrella. We automate the process of detecting duplicated web requests by looking at a compressed signal from system events, specifically function calls and returns.

8 Conclusion

In this paper, we have presented a systematic method and an automated tool called GRIFFIN for detecting an important problem that afflicts many web servers, namely, duplicate client browser requests. This causes an artificially high load on servers and corrupts server and client state. Culling together many blog posts and developer forum reports, we identify the two fundamental root causes of the problem and come up with a solution that handles both, without needing special case logic for the two root causes or for different browsers. We use GRIFFIN for detecting the problem in a production web portal for an NSF center at Purdue and identify that the problem is more widespread than previously identified. Our evaluation on the production site revealed no false positive. The dynamic system tracing using SYSTEMTAP is lightweight and the detection latency small enough (less than half a minute) as to be useful in practice. Our contributions were considered significant enough that the problem was fixed in the web portal and our addition to the dynamic tracing facility was accepted in its official release.

References

[1] Alexa Internet, Inc. `http://www.alexa.com/`.

[2] Apache MPM prefork. `http://httpd.apache.org/docs/2.2/mod/prefork.html`.

[3] Empty image src can destroy your site. `http://www.nczonline.net/blog/2009/11/30/empty-image-src-can-destroy-your-site/`.

[4] Empty SRC And URL() Values Can Cause Duplicate Page Requests. http://www.bennadel.com/blog /2236-Empty-SRC-And-URL-Values-Can-Cause -Duplicate-Page-Requests.htm.

[5] Heroku. `https://www.heroku.com/`.

[6] HTML 4.01 Specification. `http://www.w3.org/TR/html4/`.

[7] HUBzero powered websites. `https://hubzero.org/sites`.

[8] Scale your Applications on the Web. `https://www.openshift.com/developers/scaling`.

[9] Systemtap call-depth feature request. `https://sourceware.org/bugzilla/show_bug.cgi?id=16472`.

[10] BODIK, P., GOLDSZMIDT, M., FOX, A., WOODARD, D. B., AND ANDERSEN, H. Fingerprinting the datacenter: Automated classification of performance crises. In *Proceedings of the 5th European Conference on Computer Systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 111–124.

[11] BORDAWEKAR, R., BONDHUGULA, U., AND RAO, R. Believe it or not!: mult-core cpus can match gpu performance for a flop-intensive application! In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), ACM, pp. 537–538.

[12] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 2–2.

[13] CERF, V., AND KAHN, R. A protocol for packet network intercommunication. *Communications, IEEE Transactions on 22*, 5 (May 1974), 637–648.

[14] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 2:1–2:49.

[15] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 105–118.

[16] COSKUN, B., AND GIURA, P. Mitigating sms spam by online detection of repetitive near-duplicate messages. In *Communications (ICC), 2012 IEEE International Conference on* (June 2012), pp. 999–1004.

[17] EIGLER, F. C., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J., AND CHEN, B. Architecture of systemtap: a linux trace/probe tool.

[18] FU, Q., LOU, J.-G., LIN, Q.-W., DING, R., ZHANG, D., YE, Z., AND XIE, T. Performance issue diagnosis for online service systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on* (2012), IEEE, pp. 273–278.

[19] HACKER, T. J., EIGENMANN, R., BAGCHI, S., IRFANOGLU, A., PUJOL, S., CATLIN, A., AND RATHJE, E. The neeshub cyberinfrastructure for earthquake engineering. *Computing in Science and Engg. 13*, 4 (July 2011), 67–78.

[20] HIMMEL, M., HOFFMAN, R., AND MALL, M. System and method for preventing duplicate transactions in an internet browser/internet server environment, May 22 2001. US Patent 6,237,035.

[21] HUBER, N., HROMALIK-POUCHET, M. S., CAROZZI, T. D., GOUGH, M. P., AND BUCKLEY, A. M. Parallel processing speed increase of the one-bit auto-correlation function in hardware. *Microprocess. Microsyst. 35*, 3 (May 2011), 297–307.

[22] KIRK, D. B., AND HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[23] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[24] MCLENNAN, M., AND KENNELL, R. Hubzero: A platform for dissemination and collaboration in computational science and engineering. *Computing in Science & Engineering 12*, 2 (2010), 48–53.

[25] MOGUL, J., AND MOGUL, J. C. Trace-based analysis of duplicate suppression in http. `http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-99-2.html`, 1999.

[26] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[27] SOUDERS, S. High-performance web sites. *Commun. ACM 51*, 12 (Dec. 2008), 36–41.

[28] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM Comput. Commun. Rev. 30*, 4 (Aug. 2000), 87–95.

[29] STEVE W. Monkey Code. `http://code.alittlegoofy.com/2008/12/i-found-somethng-peculiar-about.html`.

[30] VALLÉS, E., AND ROSSO, P. Detection of near-duplicate user generated contents: The sms spam collection. In *Proceedings of the 3rd International Workshop on Search and Mining User-generated Contents* (New York, NY, USA, 2011), SMUC '11, ACM, pp. 27–34.

[31] VENABLES, W. N., AND RIPLEY, B. D. *Modern Applied Statistics with S*. Springer Publishing Company, Incorporated, 2010.

[32] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 117–132.