MT-WAVE: Profiling Multi-Tier Web Applications

Anthony Arkles
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, CANADA
S7N 5C9
aja042@cs.usask.ca

Dwight Makaroff
Department of Computer Science
University of Saskatchewan
Saskatoon, SK, CANADA
S7N 5C9
makaroff@cs.usask.ca

ABSTRACT

Modern web applications consist of many distinct services that collaborate to provide the full application functionality. To improve application performance, developers need to be able to identify the root cause of performance problems; identifying and fixing performance problems in these distributed, heterogeneous applications can be very difficult. As web applications become more complicated, the number of systems involved will continue to grow and full-system performance tuning will become more difficult.

We postulate that multi-tier profiling, starting at the web browser, is the appropriate way to solve this problem. Instrumenting from the web browser, as the user experiences it, ensures that we can tell what each service in the application is contributing to overall page-load time; thus, each tier must provide instrumentation data that developers can use to quickly identify the root cause of performance problems.

We have built MT-WAVE, a system that integrates with the different tiers of a web application (including a browser extension) and collects light-weight instrumentation to a central location via X-Trace [13] facilities. The collected data is presented with our visualization system that provides varying levels of detail. To validate our approach, we performed case studies of two applications, both showing performance insight. In particular, we identified and fixed a significant and unintuitive bottleneck in an open-source project management application and verified caching behaviour in a cloud-hosted commercial product. While specific technologies are used in our case study, we believe that most web technologies in common use today would require straightforward modifications to be able to utilize MT-WAVE tracing facilities.

This tool is designed to be used by application developers and system administrators while testing new software, or after deployment when it becomes clear that existing performance is not meeting user needs.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Measurement Techniques; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Measurement, Performance

Keywords

web applications, performance tracing, visualization

1. INTRODUCTION

Modern web applications consist of many independent services that collaborate to provide the full application functionality. These services often run on different machines, sometimes even hosted in separate administrative domains (e.g. cloud hosting or outsourced services). Further, web applications often have significant client-side code that runs in the user's browser.

This paper describes the problems associated with performance measurement in these distributed systems and a technique for aggregating performance data from distributed systems. The system also includes a framework for data extraction and processing, with a sample hierarchical visualization system that provides varying levels of detail (so the system analyst can switch between high- and low-level views of system performance).

Several authors have identified the need for full end-to-end multi-layer tracing. Endo *et al.* [9] claim throughput-based measurement of system performance does not accurately capture the user-perceived performance of a system, explaining that throughput measurements often represent a heavy workload that does not model user behaviour accurately (for example, a high volume of events very quickly in time) and does not accurately capture the latency variation experienced from request-to-request. Fu *et al.* [14] identified the utility of distinguishing between server latency and network overhead when retrieving a web page, along with information on the actual composition of web pages. Hellerstein *et al.* [16] eloquently describe the tracing requirements as "finger tip to eyeball" performance; measuring from when the user starts the transaction to when the final results are presented. Dapper [27] is a multi-layer tracing framework in use at Google, which shares some similarity to MT-WAVE, but only performs multi-layer tracing in the back-end. For a more detailed explanation of our choice of instrumentation techniques, see the related work.

Jones [19] points out that when solving performance problems, our intuition about the relative impact of various systems is often incorrect; this corresponds to the author's industrial experience. In fact, industrial experience using some of the tools described in section 6 was the catalyst for investigating more powerful techniques for performing web application performance measurement.

The paper is organized as follows. Section 2 provides more detailed motivation and Section 3 describes the approach we've used to perform the tracing. In Section 4, we describe the data collection, storage, and visualization techniques we've used. Section 5 explains our experience using MT-WAVE on real applications. Section 6 outlines a number of different related distributed computing tracing techniques. We conclude in Section 7 including elaborations on future directions.

2. THE PROBLEM

Distributed web applications often consist of many disjoint systems; these can be written in different programming languages, exist on different servers, or within different administrative domains, and provide many different types of system performance monitoring. Despite the significant heterogeneity in these systems, a performance analyst needs to see how all of the systems behave and interact in order to understand holistic system performance.

With an understanding of overall system performance, an analyst still faces a difficult decision: which systems should be improved in order to improve overall end-user experience? Traditionally, this problem is solved by identifying system bottlenecks and investing effort to improve the bottlenecked system. In a complex web application, the concept of "a single bottleneck" is blurred; there are many different user actions, and it can be difficult to determine how an improvement in an identified bottleneck will actually improve the overall performance from a user's point-of-view.

To solve these problems, the analyst needs access to a very broad set of data to determine which resources are used most often, which systems are involved in the use of those resources, how long each of the involved systems takes to perform data processing, and which data processing, if any, happens in parallel, etc. Currently, there are tools that provide portions of this data, but do so independently of the other systems involved.

To collect and present this data to an analyst, we have developed MT-WAVE (Multi-Tier Web Application Visualization and Evaluation), which addresses the challenges in large-scale web application performance monitoring and analysis. The main characteristics of MT-WAVE are the following:

- An existing distributed data collection service, X-Trace [13], has been retrofitted with extensions that assist with collecting web-based profile data. We have developed a report collection and retrieval facility that communicates using JSON [8] for simple JavaScript integration.
- We implemented a client-side tracing system that associates all of the requests required to serve a web page with unique task and operation identifiers, while gathering user-perceived timing information.
- We developed lightweight data collection modules for Django[1] and Google App Engine[2] that instrument and forward X-Trace metadata to other services called from the application.
- We provide a data visualization system that interacts with the JSON X-Trace interface to provide analysts with variable high- and low-level system views to determine which system components are likely to be causing the highest-impact performance problems.
- We use MT-WAVE to investigate the page load performance of two sample web applications.

To put the problem in concrete terms, we will consider two case studies that show the usefulness of MT-WAVE. First, we instrumented and evaluated Basie [4] (section 5.1), a student-developed "web-based software project forge" for managing software development. It presents a web-based interface to version control systems, project documentation, (via a Wiki), and an issue tracker (for keeping track of bugs and features that need to be implemented in a software system). Basie is built using Python and Django, using an SQLite database and SVN for version control integration. Second, we examine MashedIn (section 5.2), a multi-tiered application that runs on the Google App Engine. The MashedIn application aggregates data from different social networks (Twitter, Facebook, etc) and combines the results to help users find people they may know. Architecturally, this application makes many calls to third-party services and relies heavily on the App Engine DataStore and Memcache services to improve system performance.

To get a better sense of MT-WAVE deployment, we consider a request for a Dashboard that displays recent project activity and the services that will be used. The browser first makes a request for "/basie/project/dashboard", which the web server directs to the Basie application. To display all the information on the dashboard page, the application needs to query the SVN repository (to find recent commits), along with a number of database queries to determine which milestones there are, how many tickets there are for the project (open/total), which Wiki pages have been recently modified, what recent activity there has been, which user is currently logged in and which tickets are assigned to them, etc. After receiving this first response, the browser loads other resources (images, CSS, JavaScript, etc). The overall page load requires 13 total HTTP requests from the browser and 65 database queries. If this page were performing slowly, where would a performance analyst start? There are many different components to the page load/rendering process; getting a clear big-picture view of the system is necessary to begin the debugging process.

3. MULTI-TIER PERFORMANCE TRACING

When a web browser makes a request for a page, it sets into motion a large number of activities. This request eventually reaches a web server that will process the request and return a response. This response often includes references to other resources (for example, the images, CSS, and JavaScript content required to display the web page). Each of these requests may required additional server-side

[1] `http://www.djangoproject.com/`
[2] `http://code.google.com/appengine/`

processing before a response can be generated, and these responses may lead to further resource requests. Additionally, when the user clicks an element in the document, there may be additional requests generated (even though a new page is not loaded; for example, consider AJAX [25]). All of these requests are considered to be part of the page response time and can be investigated using MT-WAVE.

As web applications become more complicated, the number of systems involved will continue to grow. Traditionally, a web application often consisted of a basic application (written in a high-level language like Python, PHP, or Java) and a database server.

Cloud-hosted applications have a different environment to work in that traditional web applications. Google App Engine, for example, provides the DataStore, Memcache, UrlFetch, Offline Task Queue, Image Processing service, and several other services. On top of the large number of different services provided, web applications can no longer be expected to run on any particular machine; the serving infrastructure dynamically assigns machines to run applications. The large number of services and dynamic serving behaviour make it much more challenging to predict how an application will perform.

For MT-WAVE to trace web applications, each service used by the application will require source code modification. Fonseca *et al.* [12] say that changing applications for X-Trace instrumentation is straightforward and has minimal impact; our experiences agree. An alternative approach to this would be configuration-based tracing; we've chosen source code modification instead, because this gives us the ability to capture richer application-specifc data. We've found that in most cases, the modifications required are quite small. When the application cannot be directly modified, it is often possible to wrap the application in a thin middleware layer that treats the component as a black box.

3.1 X-Trace Summary

While a full treatment of the X-Trace system is available elsewhere [13], it is beneficial to refer to the important parts of it used in the MT-WAVE system. X-Trace fundamentally consist of two concepts: Tasks and Operations.

An X-Trace Task is the container for an entire "large-scale" capture; within the context of MT-WAVE, a Task is associated with the entire page a user is trying to view. Tasks are identified by a random 8-byte identifier and have a one-to-many relationship with X-Trace Operations.

X-Trace Operations are the individual events that make up an X-Trace Task. Each operation has an associated *OpId* (which is also a random 8-byte identifier), a *timestamp*, an *Agent* field (to indicate which system is logging the event), a *Label* field (to indicate what is actually being logged), an *Edge* field, and can contain other fields that the logger adds.

The Edge field is the important part for establishing causality. Each operation is annotated with the OpId of its causal parent; using these edges, we can reconstruct the causality chain to determine which events "caused" other events. The resulting Event Tree contains all of the causality and timing information of the events for the entire Task.

While handling each of these incoming requests, the web application will often need to call other services; this process can happen recursively, resulting in a chain of requests. To properly understand the system interactions, it is necessary to consider this causality chain. To capture the causality, we take advantage of the X-Trace system; each full page request has an associated Task ID, each operation has an Operation ID, and each piece of logged data has an Edge back to the parent operation event in the causality chain (which eventually leads back to the initial request in the web browser). The result is an Event Tree that links every recorded event with its causal parent.

3.2 Client-Side Tracing

Since we are concerned about "user perceived" system performance, the tracing must start within the web browser on the client side. We have implemented this as a Firefox extension. The MT-WAVE extension is responsible for setting the correct TaskID and OpId for outgoing HTTP requests, measuring specific events during the page load and JavaScript execution, and reporting this data back to the running X-Trace server.

The measurement portion of MT-WAVE in the web browser depends heavily on two Firefox instrumentation systems: the "observer-service" [24] and the "http-activity-distributor" [23]. These two services provide us with timestamps for the following events:

- request is queued and its response is received.
- beginning of DNS resolution.
- TCP socket connection beginning/ establishment.
- each HTTP header is successfully sent.
- browser begins to receive an HTTP response.

These services also provide us with the ability to detect when a new page is loaded, associate HTTP connections with page loads, and add custom HTTP headers to the connection. When a new page is loaded, we generate a new X-Trace Task ID and associate that metadata with the new page. Any HTTP requests that happen with that page are annotated with X-Trace headers that indicate the shared Task ID and unique OpIds that are later used to identify the causality chains.

3.3 Web Application Tracing

Once a request reaches the HTTP server, it is passed on to the web application for processing. Django applications are structured so that a request passes through a series of Middleware layers and then are processed using "view functions". Since Django was the first application environment we instrumented, we explain it in more detail. Other platforms are structured in a similar fashion, so we leveraged our experience with Django when developing other plugins.

Once a request reaches the HTTP server, it is passed on to the web application for processing. This web application could be fairly simple, or it could involve substantial computations and even be deployed on a separate application server. Thus, at the back end, there could be significant network communication between the web server, application server and database server.

Django applications pass requests through a series of middleware layers and then process them using "view functions". To instrument a Django application, we've created a custom Django Middleware layer that captures the X-Trace header sent by the Firefox extension and sets up tracing state for the view function and all underlying services. Specifically, this extracts X-Trace Task and OpIds from the incoming request and creates a stack to hold further OpIds as the request propagates through function calls (to maintain the causality chain as methods are invoked and events are logged).

Currently, the Django tracing framework only logs events explicitly; in the future, we plan to automatically invoke the Python cProfile system [26] to capture more detailed information about the application behaviour. The event logging, as implemented, uses Python decorators [28] to provide a lightweight instrumentation approach.

3.4 Database Tracing

Database tracing is currently implemented at the Django layer (wrapping the Django database call). This provided sufficient information to diagnose the query problem in Basie, but may not provide sufficient information to diagnose more subtle database problems. Although the Basie example below had a clear and easy fix, it's possible to imagine cases where fixing a slow database query can be more challenging.

In the future, we would like to actually instrument the database to provide more detailed information about query behaviour. Right now, the information we collect from the database is minimal: we only measure the latency of each query; i.e. we answer the question "Is this query slow?" If the database were instrumented, we could gain more insight into the question "Why is this query slow?" Most databases already have instrumentation hooks available; we would just need to add X-Trace event logging to these hooks.

4. DATA STORAGE, RETRIEVAL, AND VISUALIZATION

There are a variety of requirements for the logging infrastructure needed at different tiers in the serving framework. For example, the Django backend should have minimal added latency when logging request messages, while latency is less of a concern for the Firefox logging infrastructure (which can log asynchronously). The Firefox logger, though, is constrained by the types of connections that a browser is able to make.

For low-latency data logging, a simple TCP protocol is used. Essentially, the X-Trace event is assembled as a text string and sent in a TCP stream to localhost. Each serving machine runs an X-Trace proxy which receives these TCP packets and forwards them to the centralized X-Trace server; this means that the web application does not need to make external connections to any machine other than another process running on the same machine. To further improve the performance, we are currently investigating the use of Unix Domain sockets instead of TCP.

Firefox, unfortunately, is limited to making HTTP connections for logging. To accommodate this, an HTTP+JSON report service was developed. Using this service, the Firefox extension records the events to report and sends an HTTP POST request with all of the event detail directly to the X-Trace HTTP server, where the events are processed and combined with the other TCP messages from the backend. Since each event is explicitly timestamped and has a proper Edge reference, the order the X-Trace server receives the messages is irrelevant because it has all of the information required to reconstruct the time-ascending causality chains.

Google App Engine, like Firefox, is also limited to outgoing HTTP requests; an application cannot make arbitrary outgoing TCP connections, nor is the developer permitted to install the X-Trace proxy on the serving machines. To reduce MT-WAVE profiling overhead, we use the Task Queue service; this allows us to collect the trace data and send it to the X-Trace JSON-over-HTTP report service asynchronously, after the request has completed.

4.1 Django Implementation

The TCP backend for X-Trace came with the package. For Python/Django to log X-Trace reports to this, we wrote a Django "app"[3], consisting of two classes and a few helper functions:

XTraceContext: parses X-Trace metadata strings (splitting them into the flags, TaskId and OpId fields), builds X-Trace reports, and assembles them into a flattened string to send over TCP.

XTraceStack: keeps a stack of nested XTraceContext objects to properly nest the causal chains of events. For thread safety, this is stored in thread-local storage.

As the correct state is configured when each incoming request is processed by our Django Middleware layer, an entire application can have access to the instrumentation layer by adding a single line of configuration code.

4.2 X-Trace JSON Implementation

To modify the X-Trace infrastructure to allow for JSON-over-HTTP event reports, we added two classes: JsonReportSource and JsonReportHandler. JsonReportSource implements X-Trace's ReportSource interface, which allows it to add received reports to the general report stream with the use of an HTTP server. An instance of JsonReportHandler receives the POSTed reports. When a client sends a properly formatted POST request to the HTTP server started by JsonReportSource, JsonReportHandler extracts the X-Trace report in JSON format, decodes it into the native X-Trace format, and puts it into the general X-Trace report stream.

To simplify the external client's requirements for using the JsonReportSource, the format of the POST data closely mirrors the X-Trace event format. While the standard X-Trace event consists of series of "Key : Value" pairs, the JSON event format consists of a JSON Object that has the same set of value pairs. See Figure 1 for a side-by-side comparison.

The advantage to using this format is that many clients can easily create it using JSON tools; for example, Firefox has a native JSON encoder ("@mozilla.org/dom/json;1") that will take a JavaScript object and return it in the JSON string format. This saves us from having to build up a large string, worrying about character encoding and escaping, etc.

4.3 Data Retrieval

The Data Retrieval system must accommodate the restrictions caused by operating in a web browser. The existing X-Trace system returns the recorded events in a large text stream, with the expectation that the receiver is capable of parsing and manipulating this data.

While it is possible to perform large-scale data manipulation using JavaScript in a web browser, this is incredibly inconvenient. As an alternative to doing this, we've implemented a JSONP-over-HTTP [17] event and report retrieval

[3]Think of a Django app as a library, rather than an application.

```
X–Trace Report ver 1.0
X–Trace: 190fe0c63c83f39864704fabb793d81045
  Agent: HttpRequestObserver
  Label: Page request for: http://lui...
  Edge: 0000000000000000
  Epoch: 1.269842740663E9
  Timestamp: 1.269842740663E9
```

(a) Standard text-based format

```
{ "X–Trace" : "190fe0c63c83f39864704fabb793d81045",
  "Agent" : "HttpRequestObserver",
  "Label" : "Page request for: http://lui...",
  "Edge" : "0000000000000000",
  "Epoch" : "1.269842740663E9",
  "Timestamp": "1.269842740663E9" }
```

(b) JSON format

Figure 1: Comparison of X-Trace Event formats

system. This allows JavaScript code to retrieve the recorded events easily, providing them in a format that is ready to use by the analysis and visualization tools. The data returned is formatted in JSON form for further processing by the visualization framework.

To provide access to the data, a "Viz" option is added to the standard X-Trace Reports list. When clicked, this redirects the user to the Data Visualization system (section 4.4). To load the X-Trace Reports, the data visualization system GETs JSON resources in the form "/traces/{taskId}". These resources are formatted similarly to the JSON format in the Firefox plugin.

4.4 Visualization

This system collects a large volume of data; for the sample application, a single page request generates around 400 events per request, with some requests generating several thousand events (when experimenting with Basie, we encountered page requests with more than 10,000 events). Organizing this data into meaningful information requires that the data be divided into high- and low-level views.

At the high level, each requested resource is displayed with bounds indicating the start and end time. This allows the analyst to see how long each resource takes to load, along with the time-ordering of requests. These bounds are overlayed with a summary of the events happening inside them; we can see at a glance how much of the total request time is occupied by back-end processing.

Once an analyst has identified a candidate for speed improvement, clicking on the candidate event zooms in and shows all of the causal relationships between the recorded events on a single timeline. This allows the analyst to focus on the specific events that are likely to point to the bottleneck for that resource. For an example, we trace the Basie Dashboard page load from section 2.

The high-level trace view (Figure 2) shows a timeline of the entire page load. Often, page loads consist of batches of requests with large time gaps between them (for example, JavaScript code often requests additional resources via AJAX). By modifying the visible timeline (zooming and scrolling), the analyst can easily vary the level of detail required to understand the overall trace behaviour. When a user moves a mouse pointer over one of the HTTP requests, the right-hand side displays request summary information.

Once the analyst has chosen a particular HTTP request, clicking on it displays the low-level request details view (Figure 3); this view shows every X-Trace event that was logged while processing the request. In this view, you can see exactly how long is spent processing in each tier (and each sub-function in those tiers). Events are ordered left-to-right based on real time, and top-to-bottom based on causal relationships (events at the bottom are "leaf events").

5. EXPERIENCE / CASE STUDIES

5.1 Diagnosing and Fixing Problems: Basie

To demonstrate a real deployment of MT-WAVE, we step through the process of fixing a performance problem in Basie. While using the software, we noticed that the Tickets page seemed to get slower and slower as more tickets were added to a project. We add the Django middleware to the Basie project and add a few trace points to general places in the application (SQL queries and commits, view invocation, middleware invocation, etc). Each one of the instrumentation points only requires a single line of code.

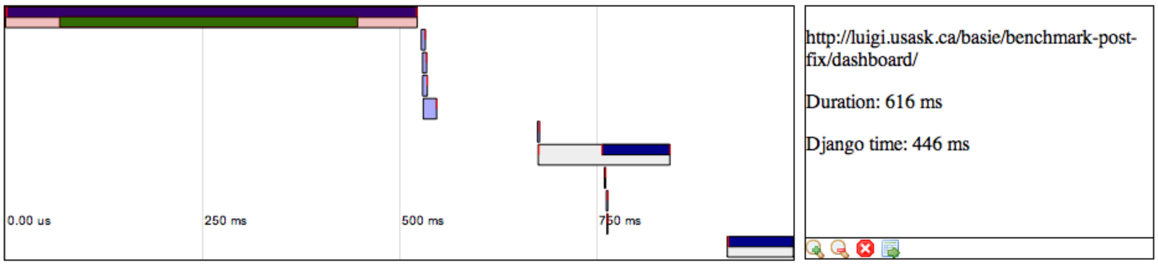
Figure 4 shows the high-level view of the Ticket page; in this view, we see that the initial page request takes a long time to execute. The initial request takes 6 seconds, almost all of which is spent doing backend processing. This is an ideal candidate for optimization. Clicking on this page request leads to Figure 5, which shows the low-level view; in the low-level view, the thick black bar is actually a large number of short database queries but we are sufficiently zoomed out that they do not appear as obvious individual events. Later on, we will examine those particular events.

Now that it is obvious that there was a problem with the ticket system, we measured the performance of the Tickets page as tickets are added. To automate this process, we used Selenium[4], a Firefox extension to generate a series of requests from the browser and render the page views. This is similar to a web-crawler, but includes the additional activity of the page display. In a loop, we added a ticket to the system and then measured the page load time. Figure 6 shows the results of this. Clearly the system performance slows as tickets are added. We now had a test suite that we can re-run after fixing the software to determine whether or not we have actually improved system performance.

To attempt to fix the ticket page load process, we zoom in on one of the slow event traces to try to figure out what is going on. Figure 7 is zoomed in on the thick black bar. Looking at this, it becomes obvious that there are many SQL queries happening in the "Rendering HTML" stage of the page request; this is a place where we do not expect queries to happen (they should all happen in the view function before we attempt to generate HTML). We counted the number of SQL queries per page load and compared against the number of tickets in the system. The number of queries increases linearly as tickets are added.

Looking at the HTML template (and using a bit of domain knowledge gained from using Django in the past), we see that the SQL queries are a side-effect of Django's Object-Relational Mapping (ORM) library; for each ticket, the template is displaying information about the creator of

[4]`http://seleniumhq.org/`

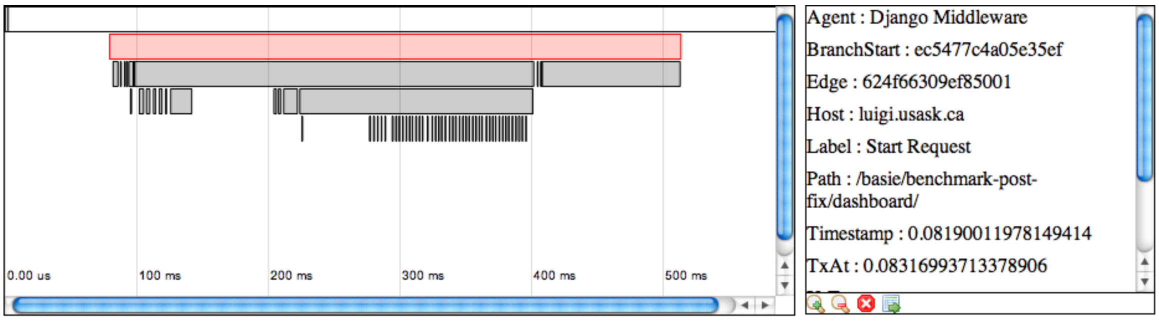


http://luigi.usask.ca/basie/benchmark-post-fix/dashboard/

Duration: 616 ms

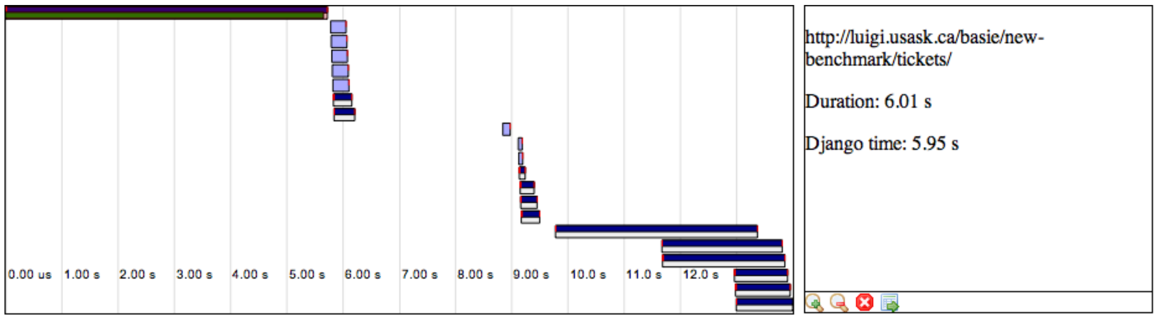
Django time: 446 ms

Figure 2: MT-WAVE showing the high-level view of a page load for the Basie Dashboard.



Agent : Django Middleware
BranchStart : ec5477c4a05e35ef
Edge : 624f66309ef85001
Host : luigi.usask.ca
Label : Start Request
Path : /basie/benchmark-post-fix/dashboard/
Timestamp : 0.08190011978149414
TxAt : 0.08316993713378906

Figure 3: Low-level view of the HTTP request from Figure 2.

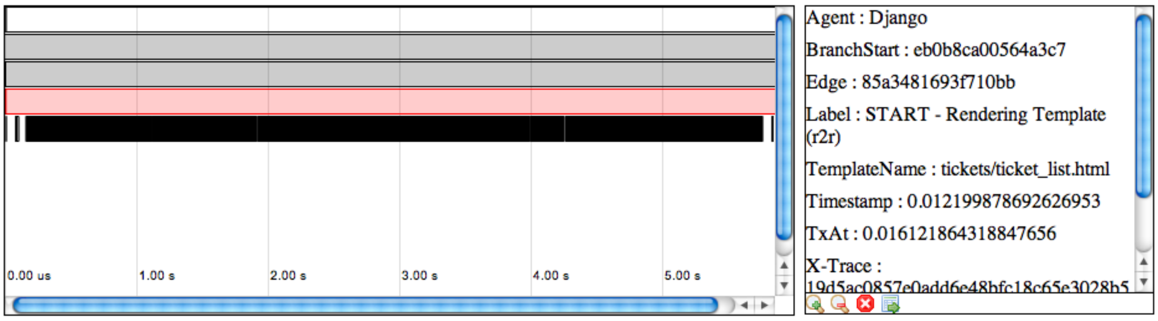


http://luigi.usask.ca/basie/new-benchmark/tickets/

Duration: 6.01 s

Django time: 5.95 s

Figure 4: High-level view of the Basie Ticket page.



Agent : Django
BranchStart : eb0b8ca00564a3c7
Edge : 85a3481693f710bb
Label : START - Rendering Template (r2r)
TemplateName : tickets/ticket_list.html
Timestamp : 0.012199878692626953
TxAt : 0.016121864318847656
X-Trace :
19d5ac0857e0add6e48bfc18c65e3028b5

Figure 5: Low-level view of the initial request from Figure 4.

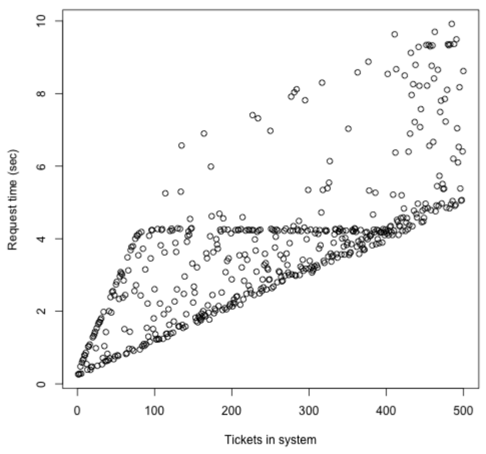


Figure 6: Request time for the Ticket View page as tickets are added to Basie.

the ticket and the milestone assigned to the ticket. Unfortunately, because the underlying data model stores these in different SQL tables, the ORM library automatically performs queries to obtain this information. Figure 8 shows the SQL summary table in MT-WAVE that highlights the queries that are being executed most frequently.

An easy way to work around this behaviour is to look up all of the milestones and users before rendering the template and filling in this data in the view function of the application; this means that we only execute one query to get all of the users and one query to get all of the milestones. Before making this fix, we ran another Selenium script to retrieve the ticket page twenty times (with a fixed number of tickets) to get a feel for the request time distribution.

We then made the "one query" fix to the view function and load the page in Firefox with MT-WAVE enabled; the first trace shows significantly fewer SQL queries than the original trace did. To verify that the performance has improved, we re-ran the 20-page-loads experiment and compare the results. Figure 9 shows the before-and-after behaviour.

Finally, to validate that the fix works in the general case, we re-ran the "add tickets and measure page load time" experiment from the beginning. Figure 10 shows the results; not only is the new system faster, but the slope of the line is much smaller–the system should scale to a much larger number of tickets before it starts to get slow. To verify that the number of SQL queries has been reduced, Table 1 shows the original SQL queries and the new SQL queries–there is now a constant number of queries, even though the number of tickets are growing. The original code was making 3 database queries per ticket.

5.2 Confirming Solutions: MashedIn

After obtaining confirmation that this technique could be valuable in identifying and visualizing the time spent in rendering web pages, we instrumented MashedIn[5], an application developed by a local software company that integrates with several third-party services and executes on Google App Engine.

There had been extensive observations of performance difficulties with this application. When we arrived, many of these performance problems had been solved by using some of the tools mentioned in Section 6 and many hours of trial-and-error with adhoc logging statements sprinkled throughout the code base. As we started using the tool, a developer remarked "I wish you'd have been here a few weeks ago, this would have really helped solve a problem we were having."

[5]`http://www.mashedin.com/`

Tickets	Queries/Request	
	Before	After
1	37	36
100	334	36
200	634	36
300	934	36
400	1234	36
500	1534	36

Table 1: Comparison of SQL queries/request between the original Basie code and the modified Basie code

To return results quickly, the MashedIn application relies heavily on application-level caching through the Memcache service. We decided to verify the effectiveness of the caching behaviour using MT-WAVE.

Figure 11 shows a section of the high-level output for a page in the MashedIn application. The highlighted request is a key AJAX request in the page load. The requests in the "blocked requests" section of the trace are created by JavaScript code that operates on the results from this AJAX request; until this first request in finished, the browser does not know what other content it will need to load. By decreasing the time required to process this request, we can improve the overall page load time. Traditional back-end profiling tools would not have identified that this request was part of the critical path for page load times, and traditional front-end profiling tools would not have been able to explain *why* this request took so long.

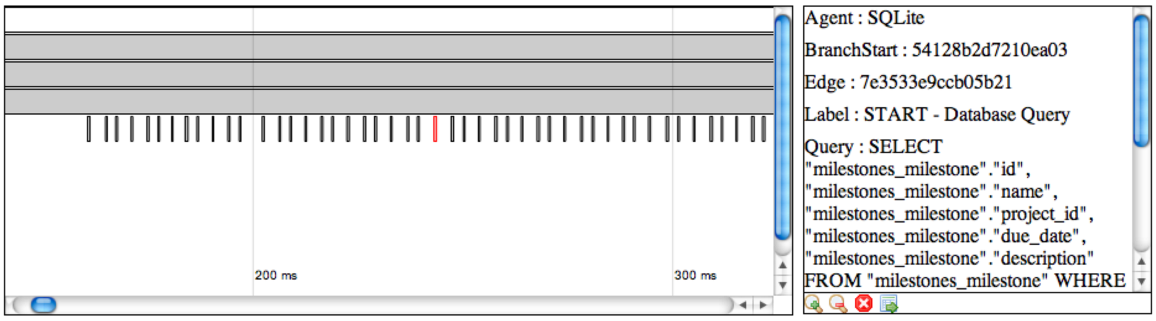
Figure 12 shows the detailed traces obtained from this request; the trace on the left shows the AJAX request with a cache miss, the trace on the right shows a subsequent AJAX request with a cache hit. Effective caching policies significantly reduce the overall load time (from $8.9s$ to $3.4s$) by caching the results from request (c). The results from request (a), though, do not appear to be cached between page requests; modifying the software to cache the results from this request could further improve the load time.

While the developers of MashedIn did not have an immediate performance problem to debug while we were working together, the results provided clear confirmation that their caching approach was effective in decreasing page load times and provided insight into further performance enhancements.

5.3 Automated Tracing: Basie

In the Basie case study, we alluded to automated data collection; here, we elaborate on the mechanisms used and provide some example use cases. Using Selenium, we set up a series of browser actions that run in an automated fashion. To do periodic instrumentation, we also set up a loop and run the script with a delay; all of the MT-WAVE traces are saved and are available for post-processing.

A use case for this arose while we were testing Basie; depending on the time of day we were testing it, we would get significantly different network latency measurements. This makes sense, of course, because we were doing the testing



Agent : SQLite

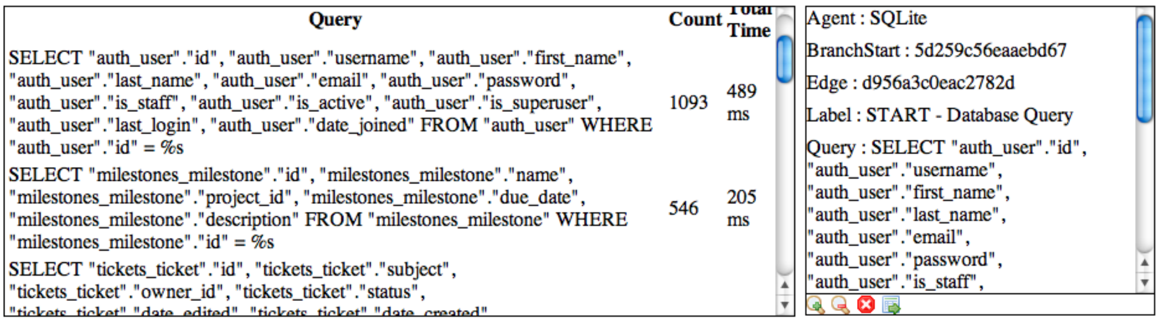
BranchStart : 54128b2d7210ea03

Edge : 7e3533e9ccb05b21

Label : START - Database Query

Query : SELECT "milestones_milestone"."id", "milestones_milestone"."name", "milestones_milestone"."project_id", "milestones_milestone"."due_date", "milestones_milestone"."description" FROM "milestones_milestone" WHERE

Figure 7: Low-level view of the initial request from Figure 4, zoomed in on the thick black bar from Figure 5.



Query	Count	Total Time
SELECT "auth_user"."id", "auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email", "auth_user"."password", "auth_user"."is_staff", "auth_user"."is_active", "auth_user"."is_superuser", "auth_user"."last_login", "auth_user"."date_joined" FROM "auth_user" WHERE "auth_user"."id" = %s	1093	489 ms
SELECT "milestones_milestone"."id", "milestones_milestone"."name", "milestones_milestone"."project_id", "milestones_milestone"."due_date", "milestones_milestone"."description" FROM "milestones_milestone" WHERE "milestones_milestone"."id" = %s	546	205 ms
SELECT "tickets_ticket"."id", "tickets_ticket"."subject", "tickets_ticket"."owner_id", "tickets_ticket"."status", "tickets_ticket"."date_edited", "tickets_ticket"."date_created"		

Agent : SQLite

BranchStart : 5d259c56eaaebd67

Edge : d956a3c0eac2782d

Label : START - Database Query

Query : SELECT "auth_user"."id", "auth_user"."username", "auth_user"."first_name", "auth_user"."last_name", "auth_user"."email", "auth_user"."password", "auth_user"."is_staff",

Figure 8: SQL summary view of the initial request from Figure 4.

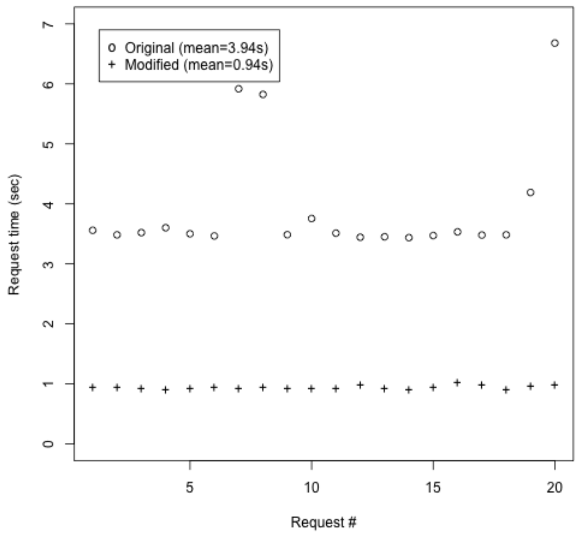


Figure 9: Before and after view of the Ticket List performance at a constant number of tickets.

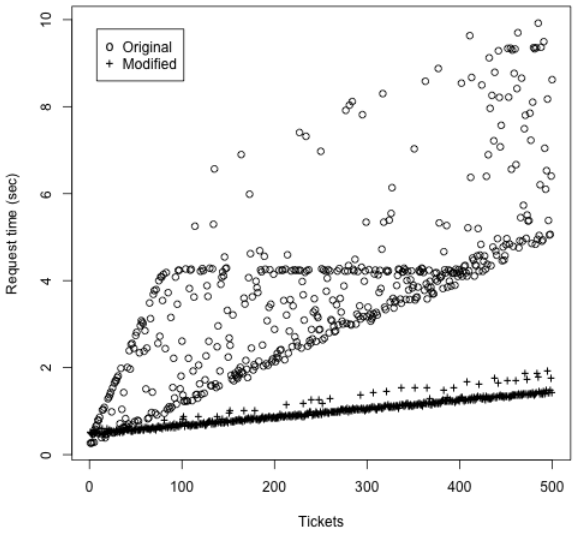
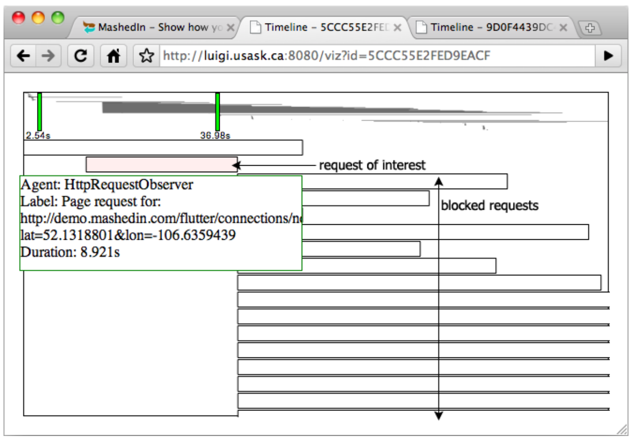


Figure 10: Total page load times: original Basie code and modified Basie code



Agent: HttpRequestObserver
Label: Page request for:
http://demo.mashedin.com/flutter/connections/n
lat=52.1318801&lon=-106.6359439
Duration: 8.921s

Figure 11: High-level view of a MashedIn page load with cold caches.

on the public University network and network latency would vary as students came and left campus (they weren't using Basie, but just adding additional load to the network).

For this example, we collected data in two batches for about an hour each. The first batch of data was collected from off-campus, and the second batch was collected on-campus during low load. Figure 13 shows the difference in latency between the two data sets. As expected, page load times are signicantly larger when network latency is involved, but back-end processing times remain fixed.

An important thing to note from Figure 13 is that traditional tools would not clearly explain the measurement differences. A back-end profiler would simply show that request times did not have significant variation (indicated by the "Django processing time" points on the plot), and a front-end profiler would show that request times had significant variation between the two measurements. Neither, though, could percisely point the finger at network latency.

MT-WAVE's JSON interface simplifies post-processing of the data. To analyze the data from this experiment, a small Python script was used to query the event reports and summarize them into comma-separated (CSV) text. Once the data is in this format, existing data processing tools can be used for analysis.

5.4 MT-WAVE Performance Overhead

Monitoring does not come for free. We have to collect traces and send them to the X-trace server. Table 2 shows the latency in web page load in the new system and in the old system. As the number of events increase, the user-perceived latency increases; for low event volumes, this overhead compares favourably with Dapper, but with higher event volumes, the added latency can become significant.

	Before	After
with MT-Wave	3.52s (4324 events)	1.05s (216 events)
without MT-Wave	2.03s	0.92s
Overhead	1.49s	0.13s
Overhead (%)	74%	14%
Overhead per event	344μs	602μs

Table 2: MT-Wave Overhead for Basie

Even in circumstances when much more tracing is required, we can tolerate a slowdown in test environments to more closely examine all the events taking place. In production deployments, a coarser view of the events can be deployed with very lightweight intensity, reducing the latency impact. Additionally, the Python tracing code would benefit from optimization; preliminary results tracing PHP code using a C module indicate that we can reduce the per-message latency significantly.

6. RELATED WORK

The inspiration for the main mechanism used in MT-Wave is X-Trace [13], a general-purpose framework for collecting causal event chains in a distributed system, which we leverage heavily for our underlying data collection. Out of the box, though, X-Trace does not provide any hooks that enable web tracing, which we've added. Early work on web tracing was done by the WebMon system [15] and integrated in a product from HP called Web Transaction Observer.

Other work on client side measurements includes Firebug[6] and FirePHP[7]. Firebug is an open-source project that integrates request tracing into the Firefox browser. Its biggest limitation is that it only works at the browser-level. Working at the browser-level is a good first step, but Firebug on its own does not provide any mechanism to understand the system holistically. As an extension to Firebug, FirePHP enables PHP applications to log events into the Firebug system. This is one step closer to understanding how the application is behaving, but only provides a facility to report log messages to a client (via HTTP response headers). FirePHP does not have any facility for instrumenting any other systems involved (e.g. databases), nor does it do any logging to a centralized log-collection facility. Another downside to FirePHP is that the (possibly untrusted) client receives all of the logging messages, which could contain sensitive data.

YSlow[8] is a tool developed at Yahoo to provide recommendations for performance improvements based on their "rules for high performance web pages". This tool also integrates with Firebug, monitoring the results that it collects and comparing the collected data with their rule set. Based on which rules are violated, advice is given to the user. Like the other tools above, YSlow is incapable of viewing anything happening beyond what the browser sees; any slowdowns that are caused by specific things happening at any layer below the HTTP requests are completely invisible.

Engineers at Google have developed Dapper [27], a multitier tool with similar goals to MT-WAVE, but significantly different implementation and usage. As they explain, most of Google's tiers used a shared RPC library, and Dapper is tightly integrated into this library. Further, Dapper only performs back-end measurements (starting at the first web server, instead of at the browser). A significant advantage of this approach is that they can collect trace data without requiring clients to install a browser plug-in. A disadvantage, though, is that the collected traces do not capture user-experienced latency, but rather the latency experienced by the front-end web server. To minimize the performance impact, they use sampling – instead of tracing every request, they only trace a random subset; fortunately, they have a

[6] `http://getfirebug.com/`
[7] `http://www.firephp.org/`
[8] `http://developer.yahoo.com/yslow/`

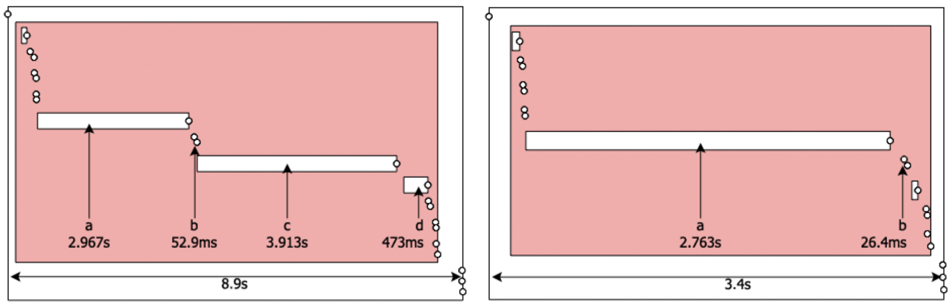


Figure 12: Detailed view of the MashedIn AJAX request from Figure 11.

high-traffic site and can get statistically-significant data by only sampling 1-10% of their requests.

Sherlock [2] is a tracing system that identifies failing components in large-scale networks. Using passive traffic sniffing, Sherlock builds an Inference Graph that contains the per-service component dependencies and a model of the network topology. When services start to fail, their Ferret algorithm calculates probabilities for each service and determines the most-likely faulty service. Sherlock, like many of the other existing work, is a network monitoring tool based on black-box service models; while it can identify a failing service, it has no insight about how to fix the failed service. For extremely complicated applications, a Sherlock-like monitoring service would serve as a good first step into solving performance problems.

cProfile [26] is the built-in Python profiling system. It intercepts function calls in a running Python interpreter and times their executions, resulting in an aggregate report showing how much time is spent in each function. This is a very powerful tool for diagnosing back-end performance problems, but has a few limitations within the context of web applications; for example, cProfile has no visibility into any activity above or below the Python level, nor is it capable of aggregating data across multiple web requests. In the future, we will likely integrate cProfile output with MT-WAVE to provide more valuable information to the analyst.

Magpie [3] performs back-end event tracing; instead of establishing global identifiers (e.g. X-Trace TaskIds), the authors use a fascinating event parsing and correlation framework to reassemble event causality chains. This system requires application-specific event schemas to be defined, which are used to try to stitch together event causality chains. A significant advantage of this system is that it doesn't require modifications to the application being measured. Unfortunately, for our specific task, this would require client-side installation of invasive tracing software. While, in general, the functionality provided by this system is very useful, many modern web applications come with the source available and are amenable to the simple modifications required by the MT-WAVE software. For applications that do not have the source available, integrating a Magpie-like system with MT-WAVE would not likely be difficult.

Aguilera *et al.* [1] provide an alternative black-box analysis system that uses RPC-tracing or signal processing to try to infer causal relationships. The RPC tracing system is based on identifying nesting within call/return pairs. The signal processing technique does not require the RPC-style call/return pairs to exist; instead it determines the probability of

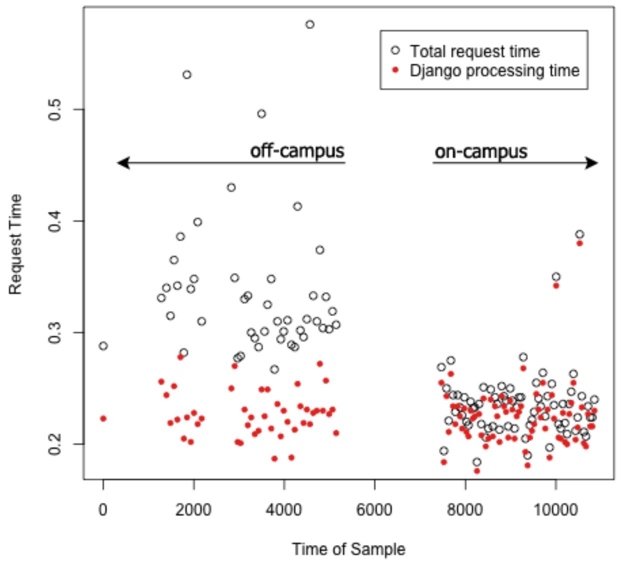


Figure 13: Comparison of page load times off-campus and on-campus.

causal relationships based on correlation analysis. Neither of these techniques are 100% accurate and rely on larger collections of data to produce significant results. These techniques could be considered for filling in gaps in MT-WAVE trace data (i.e. for subsystems that are not X-Trace enabled).

Pinpoint [7] is a system to automatically find the root-cause of system failures. This system, like MT-WAVE, is based on tagging each request with a unique identifier to trace the flow of the request through the system. While Pinpoint is not a system specifically designed for performance analysis, the trace analysis component could be very useful in conjunction with MT-WAVE.

In Stardust [29], a distributed storage system is designed from scratch with cross-machine instrumentation in mind. They use the notion of a "breadcrumb" to track a request through different systems. The breadcrumb is essentially the TaskId used in the X-Trace framework; they do not assign unique identifiers (beyond timestamp and breadcrumb) to each event, which may make reassembling the event causality chains slightly more challenging.

Feldmann's "Bi-layer HTTP and TCP" tracing system [10] performs a number of analyses based solely on packet sniffing. This system is excellent for gathering a large volume of information about traffic going to a number of different servers (for instance, when evaluating whether or not a local caching proxy would be suitable for an organization), but the coarse granularity of the system makes it a poor choice for tuning a specific application.

Whodunit [6] is an application tracing framework that captures cross-thread messages that happen over shared memory. To do this, it adds instrumentation to MOV instructions and records which pieces of shared memory they affect. By instrumenting the locks that surround these instructions, they build up a series of producer/consumer relationships between the threads in an application. This is a very powerful technique for adding instrumentation within a single executable. To track transactional flow between layers, the SEDA system (a middleware system between layers) forms queues between the systems and sets up the transaction context as items are removed from the queue. Adding and processing data from a system like this is potential future work for MT-WAVE, to perform incredibly detailed application instrumentation.

7. CONCLUSIONS/FUTURE WORK

Modern web applications are composed of a number of heterogeneous systems. From a user's point of view, though, the performance of these individual systems is irrelevant; the user experience is based entirely on the system performance as seen by the web browser. For an analyst to understand the browser-originated system performance, she needs the ability to see how the various systems that compose the application perform while handling requests. Standard industrial web application performance tracing tools have a narrow view of this performance that only provides data on each system independently.

We have built a browser extension and integrated it with a multi-tier tracing system to provide analysts with a holistic view of a web application. To demonstrate its utility, we have taken two sample application, Basie and MashedIn, and used simple instrumentation to identify precisely which components of the application add the most latency to the user's page load time. This system is still in its infancy, with many different possible avenues for future work, but we have concretely shown that it is a useful tool as-is for debugging performance problems in web applications.

Event timestamps are collected on each machine that is producing log messages; unfortunately, not every system on the Internet has perfectly synchronized clocks, and therefore the clock offsets between machines can significantly shift the event data. Fortunately, the Edges of the X-Trace reports still provide a clear causality relationship between the events, so event timing is still measured properly, but reconstructing the entire causality chain through time can be challenging. To properly synchronize the events occurring in different systems, features from Vector Clocks [11] or NTP [22] should be considered and added to the system.

Collected task data is shown in a one-task-at-a-time interface; however, it will often be beneficial to see aggregated collections of tasks to assess overall trends. The direction to go with this is still unclear; as we use the system more and receive feedback from users, a better picture should appear.

The system, as implemented, is good at collecting data and visualizing it; it does not, however, provide much assistance for automatically analyzing the data. Since we have our events structured in a tree, it would likely be possible to perform some kind of subtree isomorphism analysis on the various task traces to identify commonly used systems; once a slow, commonly-used system is identified, it can be fixed to give significant performance improvements throughout the application. Because our system operates inward from the browser, we can track exactly how much time each potential bottleneck actually contributes to the user's latency experience; this not only allows the analyst to focus their effort on the high-impact systems, but also allows them to quickly identify which systems do *not* require investigation.

The instrumentation probes provided with MT-WAVE right now are quite limited; there are many additional systems that could benefit from the added instrumentation, and if Python/Django is a good example then we can expect relatively straightforward integration with other systems. We are currently working on a PHP port, and a Java/AspectJ port is a future possibility.

A common trend in web applications is taking advantage of APIs provided by other systems, often outside of the application's administrative domain. Isaacs & Barham [18] present a case for the ability to do performance evaluation across loosely-coupled systems. The MT-WAVE framework (and X-Trace) provide most of the tools to do this already; to complete the task would require cross-AD requests to contain a host identifier where the events should be logged (along with cooperation between administrative domains). In any case, by including X-Trace headers in outgoing requests we provide the server operators in the other AD the opportunity to perform their own performance tracing within their own application, whether or not they decide to share that information.

To simplify the process of obtaining more detailed trace data (especially client-side trace data), a system like AjaxScope [20] could be useful to integrate. The main concept in AjaxScope is *dynamic instrumentation* that is added to code on-the-fly based on current demands. While AjaxScope is designed to instrument JavaScript code, the use of Aspect-Oriented Programming (AOP) (for example, AspectJ [21]) provides the ability to easily instrument system behaviour based on a set of pointcuts. AjaxScope and AOJS [30] both filter JavaScript code through a proxy that modifies the outgoing code at request-time; this allows each request to include different sets of instrumentation. This behaviour could be driven from the MT-WAVE interface, creating an interactive loop where the analyst investigates trace data, determines a component that she would like to learn more about, and then performs the request again with the new instrumentation requirements.

While AOP can help considerably with client-side code (and with server-side code), we often need to record low-level statistics on the server to determine the root cause of a performance problem. This is where integration with systems like DTrace [5] would be valuable. DTrace provides the ability to instrument user- and kernel-level code with zero overhead until instrumentation probes are added. By providing an interface to dynamically modify the set of probes, the analyst can selectively measure server-side performance metrics while minimizing the amount of overhead introduced.

Many systems that we will want to instrument have some form of internal profiling mechanism (for example, cProfile mentioned above for Python). Since we have infrastructure in place to receive log messages from these different systems, collecting and aggregating profiling information will further help the analyst understand the behaviour in the system, especially by providing specific insight as to which functions in the code are adding the significant delays to the system.

8. REFERENCES

[1] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP'03* (Bolton Landing, NY, 2003), pp. 74–89.

[2] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM '07* (Kyoto, Japan, 2007), ACM, pp. 13–24.

[3] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for Request Extraction and Workload Modelling. In *OSDI'04* (San Francisco, CA, 2004), pp. 259–272.

[4] Basie Project. A simpler software project forge. `http://basieproject.org/`.

[5] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic Instrumentation of Production Systems. In *USENIX'04* (Boston, MA, June 2004), pp. 15–28.

[6] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional Profiling for Multi-Tier Applications. In *EuroSys'07* (Lisbon, Portugal, 2007), pp. 17–30.

[7] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Intl. Conf. on Dependable Systems and Networks* (Bethesda, MD, June 2002), pp. 595–604.

[8] Crockford, D. Introducing JSON. `http://www.json.org/`.

[9] Endo, Y., Wang, Z., Chen, J. B., and Seltzer, M. Using Latency to Evaluate Interactive System Performance. In *OSDI'96* (Seattle, WA, October 1996), pp. 185–199.

[10] Feldmann, A. BLT: Bi-layer Tracing of HTTP and TCP/IP. In *9th international World Wide Web conference on Computer networks* (Amsterdam, The Netherlands, 2000), pp. 321–335.

[11] Fidge, C. J. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. *Australian Computer Science Communications 10*, 1 (February 1998), 56–66.

[12] Fonseca, R., Freedman, M., and Porter, G. Experiences with Tracing Causality in Networked Services. In *INM/WREN* (San Jose, CA, Apr. 2010).

[13] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A Pervasive Network Tracing Framework. In *NSDI'07* (Cambridge, MA, 2007), pp. 271–284.

[14] Fu, Y., Cherkasova, L., Tang, W., and Vahdat, A. EtE: Passive End-to-End Internet Service Performance Monitoring. In *USENIX'02* (Monterey, CA, June 2002), pp. 115–130.

[15] Gschwind, T., Eshghi, K., Garg, P. K., and Wurster, K. Webmon: A Performance Profiler for Web Transactions. In *WECWIS '02* (Newport Beach, CA, June 2002), IEEE Computer Society, pp. 171–177.

[16] Hellerstein, J. L., Maccabee, M. M., III, W. N. M., and Turek, J. J. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *ICDCS'99* (Austin, TX, 1999), pp. 152–162.

[17] Ippolito, B. Remote JSON - JSONP. `http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/`.

[18] Isaacs, R., and Barham, P. Performance Analysis in Loosely-Coupled Distributed Systems. In *7th Cabernet Radicals Workshop* (Bertinoro, Italy, October 2002).

[19] Jones, M. B., and Regehr, J. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *HOTOS'99* (Rio Rico, AZ, 1999), pp. 96–101.

[20] Kiciman, E., and Livshits, B. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *SOSP'07* (Stevenson, WA, October 2007), pp. 17–30.

[21] Kiczales, G., Hilsdale, E., Hugunin, J., Palm, M. K. J., and Griswold, W. G. An Overview of AspectJ. In *ECOOP'01* (Budapest, Hungary, 2001), pp. 327–353.

[22] Mills, D. L. RFC-1129: Internet Time Synchronization: the Network Time Protocol, October 1989.

[23] Mozilla Foundation. Mozilla Developer Center / Monitoring HTTP Activity. `https://developer.mozilla.org/en/Monitoring_HTTP_activity`.

[24] Mozilla Foundation. Mozilla Developer Center / Observer Notifications. `https://developer.mozilla.org/en/Observer_Notifications`.

[25] Paulson, L. Building Rich Web Applications with Ajax. In *IEEE Computer* (October 2005), vol. 38-10, pp. 14–17.

[26] Python Software Foundation. The Python Profilers. `http://docs.python.org/library/profile.html`.

[27] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. rep., Google Research, apr 2010.

[28] Smith, K. D., Jewett, J. J., Montanaro, S., and Baxter, A. PEP-318 - Decorators for Functions and Methods. `http://www.python.org/dev/peps/pep-0318/`.

[29] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. Stardust: Tracking Activity in a Distributed Storage System. In *SIGMETRICS'06* (Saint Malo, France, 2006), pp. 3–14.

[30] Washizaki, H., Kubo, A., Mizumachi, T., Eguchi, K., Fukazawa, Y., Yoshioka, N., Kanuka, H., Kodaka, T., Sugimoto, N., Nagai, Y., and Yamamoto, R. AOJS: Aspect-Oriented Javascript Programming Framework for Web Development. In *ACP4IS'09* (Charlottesville, VA, 2009), pp. 31–36.