2010

Exploration of Dynamic Web Page Partitioning for Increased Web Page Delivery Performance

Brian Michael Krupp
Cleveland State University

EXPLORATION OF DYNAMIC WEB PAGE PARTITIONING FOR INCREASED WEB PAGE DELIVERY PERFORMANCE

BRIAN M. KRUPP

BACHELOR OF SCIENCE IN COMPUTER INFORMATION SYSTEMS

Baldwin-Wallace College

May, 2005

submitted in partial fulfillment of requirement for the degree

MASTERS OF COMPUTER AND INFORMATION SCIENCE

at the

Cleveland State University

December, 2010

APPROVAL PAGE

This thesis/dissertation has been approved
for the Department of COMPUTER AND INFORMATION SCIENCE
and the College of Graduate Studies by

_____Date_____
Thesis Chairperson, Dr. Timothy Arndt
Computer and Information Science

_____Date_____
Committee Member, Dr. Ben Blake
Computer and Information Science

_____Date_____
Committee Member, Dr. Janche Sang
Computer and Information Science

EXPLORATION OF DYNAMIC WEB PAGE PARTITIONING FOR INCREASED WEB PAGE DELIVERY PERFORMANCE

BRIAN KRUPP

ABSTRACT

The increasing use of the Internet and demand for real-time information has increased the amount of dynamic content generated residing in more complex distributed environments. The performance of delivering these web pages has been improved through more traditional techniques such as caching and newer techniques such as pre-fetching. In this research, we explore the dynamic partitioning of web page content using concurrent AJAX requests to improve web page delivery performance for resource intensive synchronous web content. The focus is more on enterprise web applications that exist in an environment such that a page's data and processing is not local to one web server, rather requests are made from the page to other systems such as database, web services, and legacy systems. From these types of environments, the dynamic partitioning method can make the most performance gains by allowing the web server to run requests for partitions of a page in parallel while other systems return requested data. This differentiates from traditional uses of AJAX where traditionally

AJAX is used for a richer user experience making a web application appear to be a desktop application on the user's machine. Often these AJAX requests are also initiated by a user action such as a mouse click, key press, or used to check the server periodically for updates. In this research we studied the performance of a manually partitioned page and built a dynamic parser to perform dynamic partitioning and analyzed the performance results of two types of applications, one where most processing is local and another where processing is dependent on other systems such as database, web services and legacy systems. The results presented show that there are definite performance gains in using a partitioning scheme in a web page to deliver the web page faster to the user.

TABLE OF CONTENTS

CHAPTER I
INTRODUCTION

Web platforms continue to become the preferred platform for new and existing applications. As they continue to grow as the preferred platform, their complexity grows. This complexity is attributed to the integration of legacy and distributed applications. Where a traditional web page would mostly include static content with some dynamic content, today's web application contains more dynamic content that includes data from systems such as database servers, web services, and legacy applications including mainframe. Unfortunately, most web application server languages process web requests in a sequential matter, where a request to a distributed platform from the requested page would block the processing of the remainder of the request.

There's been much research in the area of improving web performance by caching static content and pre-fetching web content using artificial intelligence to determine which content may be loaded next. However, even with caching of static content the dynamic content of the page's performance doesn't improve. Also with pre-fetching, if the algorithm makes an incorrect decision on the future content to be

requested, resources are wasted on requesting that content and processing that content.

Our approach will utilize existing standards and protocols to partition content within a page at the source and allow the partitions of the web page to be processed in parallel to improve web page delivery performance.

CHAPTER II
RELATED WORK

Considering the impact of improving web page delivery performance has, there's been considerable research in this realm. Some of the more recent and common research in this area has been in prefetching web content and caching of static content. Caching which has been implemented in web browsers for quite some time has been coupled with proxies to allow caching to be done at an organizational level for better predictability. One hybrid method that was proposed by Huang and Hsu defined a method to mine popular surfing using a prediction-based buffer manager that resides in front of a proxy to both cache and prefetch web pages. This method combined both caching and prefetching and removes the requirement for extra software to be installed on a user's machine. (5) A different approach proposed by Pons used the Markov-Knapsack method to perform prefetching of web content by using the current web page and a Knapsack selector to determine the web objects to request. This model uses a server to keep track of prefetched pages, and pages that have been prefetched after. (13) A different approach that focuses on improving crawling performance proposed by Peng, Zhang, and Zuo looks at segmenting the web pages into relatively smaller units to

expand the reach of crawling by navigating through irrelevant content to reach more important content. This approach takes one page that may be irrelevant as a whole and divides it up to find relevancy in a particular partition. (10)

In both of the prefetching models it removes the user's machine from needing additional software, which we take a similar approach by utilizing existing protocols and standards and utilizing the web server to perform the partitioning, similar to that of the partitioning approach that is proposed above, except in the approach we propose its used to improve web page delivery performance to the user not a crawler. Also, our partitioning technique will occur at the server level, unlike the approach from Peng, Zhang, and Zuo which performs the partitioning once the page is received using the document object model(DOM) (13). Also, with our technique it eliminates any wasted resources used on predictability where there may be a missed prefetch that is never later requested.

CHAPTER III
RESEARCH ENVIRONMENT

There were many different technologies used in this research. The following contains a brief description of those technologies and why they were used.

Linux Apache MySQL PHP (LAMP)

For this research, we decided to use a Linux, Apache, MySQL, and PHP platform, also known as LAMP to research the performance of partitioning and also build the framework for performing the dynamic partitioning.

LAMP was chosen as the platform to perform the research on for several factors. It is inexpensive, it can run on most hardware, and it is also free. Also, development time in this platform would be considerably less than other enterprise platforms. In this research, we are more concerned with the ideas then the specifics of a particular language.

Specific reasons for each component of this platform are given below.

Linux

We could have used either Windows for a WAMP based platform or traditionally use Linux, and we chose Linux as we felt that we had more control over running processes and would get more accurate test results. Also for ease of automated testing, Linux would be a much easier platform to write our scripts on.

Apache

After picking Linux as the operating system, we were limited to what web servers we would be able to use. Apache is a well known web server that is easily configurable, plus it integrates well with PHP. By it being easily configurable, we could modify the number of threads quickly on the web server and analyze the impact those changes would have on testing.

Also with Apache, we could potentially extend our research in the future to include modules for dynamic caching of partitions of web pages.

MySQL

MySQL is just part of the LAMP platform. We may use it to store results of our testing so we can dynamically display graphs that show the performance gains of the partitioning of a web page.

PHP

PHP was chosen as the language to do the research in as it is the primary language for the LAMP platform, it supports Regular Expressions and XML, and the development would be faster than other traditional languages. We could develop the

framework in a more traditional language such as C or Java, however the development time would take considerably longer and we are more concerned with implementing the ideas of this research rather than the specifics of a particular language.

We will use PHPs built in XML processing for web pages that follow strict XHTML rules and regular expressions for those that don't. PHP is also well suited for parsing text which will be the primary data that we will be working with.

Perl

Perl will be used for doing some of the automated client testing by simulating a browser and making requests to the web server. For this, we will use the LWP and HTTP libraries in Perl.

AJAX

Asynchronous JavaScript and XML (AJAX) will be used heavily in this research as it will be used to make the request for partitions of web page. We will make the AJAX requests occur concurrently for a given web page through the use of closures in JavaScript, (2) this is opposed to having the requests occur sequentially or even in a traditional approach where the response of a web page is delivered all at once. The goal here is that each request gets a thread or process on the web server to process the request, and then the browser will receive the responses and put together the document.

Traditional Approach

Looking at the traditional approach first, it's a fairly simple approach, the browser requests the page, the web server receives the request, processes it in a sequential matter, and sends the response back to the browser, in this approach, no parallelism and no AJAX is involved.

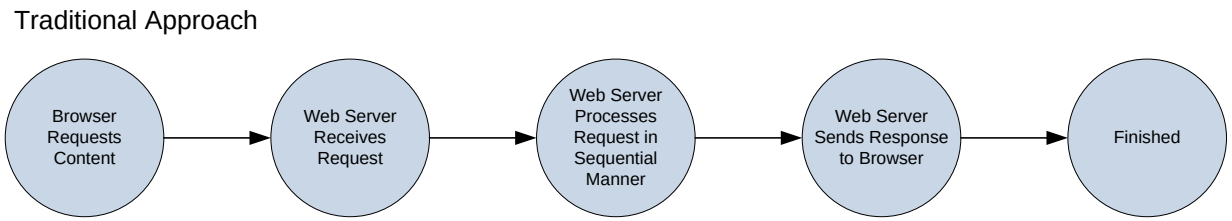

Figure 1

Sequential AJAX

Looking at sequential AJAX requests, we would make a request after a response is received, so the web server would still only be processing one request at a time, but each request is a partition of the page which is still insufficient in improving the performance of a web page:

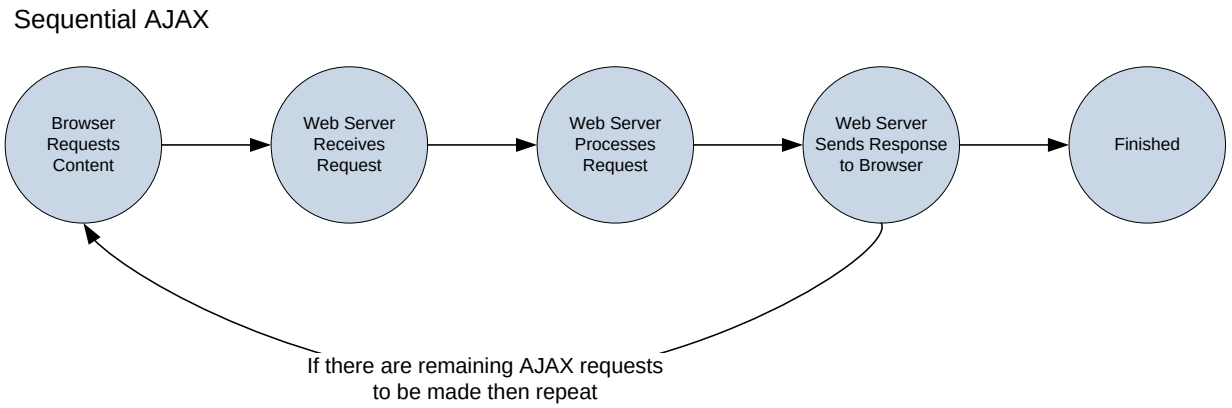

Figure 2

Let's assume that we have four partitions of the page, and that each partition takes 5ms to process, in this scenario even with the page partition each partition is processed sequentially, so the total time would take 20ms:

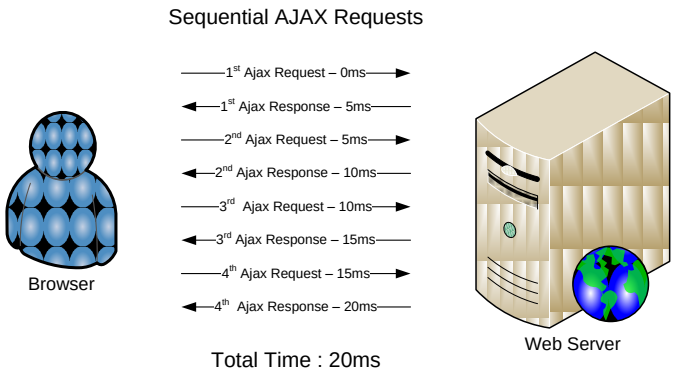

Figure 3

In this scenario, we don't have a performance improvement, more realistically it would be a performance degradation because of network latency of additional requests and additional data being sent for the each request.

Concurrent AJAX

The area where we are looking to make our gains is where a partition request requires some other system to process data such as a database query. While another system is processing the data, the web server can work on other requests until a response is received. This process would be different than previous because the browser would keep making requests for the page partitions until there are no requests left to be made, it doesn't wait until a response is received from the web server:

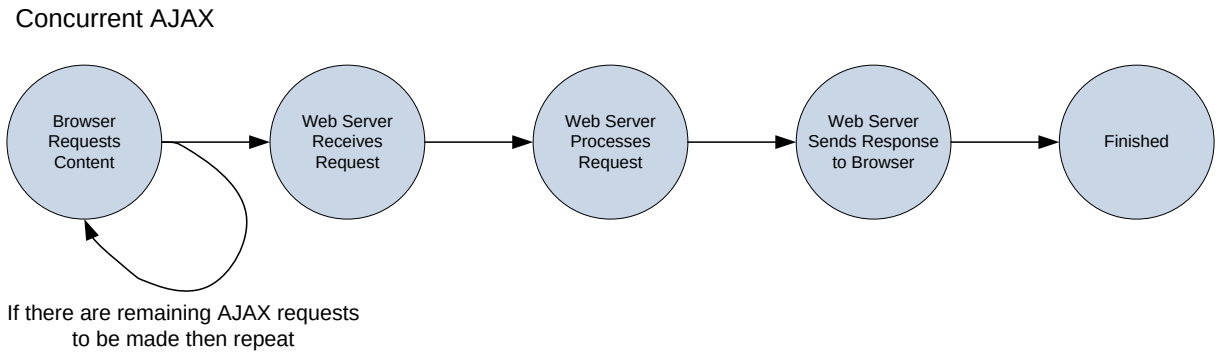

Figure 4

By partitioning our page so that it makes several concurrent requests to the web server, we allow the web server to process other requests while it may wait for a response from another service or remote machine, therefore reducing the total amount of processing time, therefore increasing performance. If we were to make the same assumption that we did in the Sequential AJAX scenario where each partition takes 5ms to process, then the time it would take to process the entire page would be reduced to the largest partition processing time:

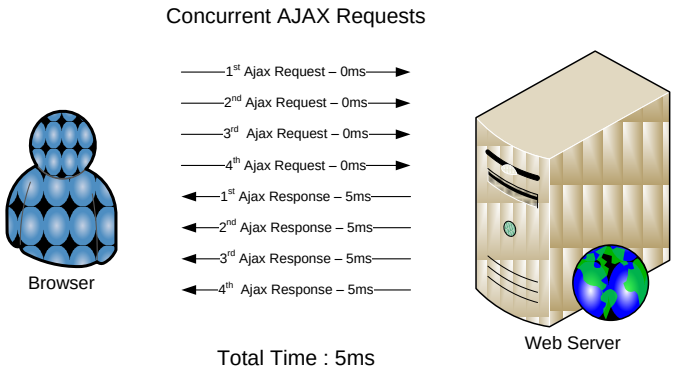

Figure 5

The best case scenario would be where each request ran on its own processor core, where it can be local or on another system. To accomplish this, we will take a page, and divide it based on a standard divider of common content of an HTML page. For this research, we will use the tag.

Summary of Research Environment

Below are the specs of the system we will use for testing our dynamic partitioning method:

Component	Dual Core Environment	Single Core Environment
CPU	Intel Core 2 Duo CPU 2.0 GHz (2 Cores to VM)	Intel Core 2 Duo CPU 2.0 GHz (1 Core to VM)
Disk Speed	5,400 RPM	
Memory	512 MB	
Network	100 MB Ethernet Connection	
OS	Ubuntu 10.4 Virtual Machine running in VMware Player 3.1.0 build-261024 on Windows 7 64 bit Host OS	
Linux	2.6.32-24	
Apache	2.2.14	
PHP	5.3.2	

Table I

CHAPTER IV
TESTING CONCURRENT AJAX SUPPORT IN MAJOR BROWSERS

To see how modern browsers will handle our concurrent AJAX Model using closures, we wrote a test script that each AJAX function will call that then sleeps for 5 seconds so this way we can determine how many calls the browser could process before queuing them up.

Mozilla Firefox 3.6

On the first test using Firefox, we noticed that it could handle 6 concurrent requests at a time. Note that the two screenshots below have AJAX responses in sets of 6:

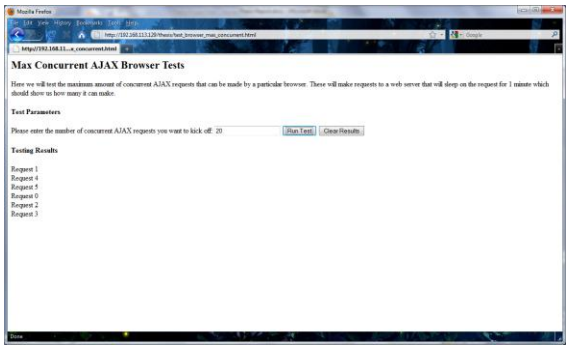

Figure 6

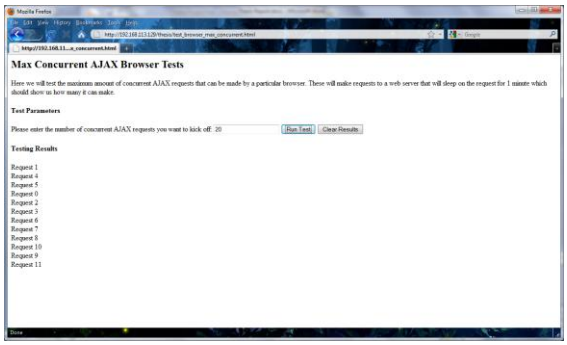

Figure 7

Internet Explorer 8.0

When running the test with Internet Explorer, we noticed the same results. However when running subsequent test, the response was cached, so we had to add some randomness to our concurrent AJAX framework (see Appendix B) to prevent that.

Also with IE, we noticed that the order of requests being responded, was more random then that of Firefox. In Firefox, the first set of requests that came back were of the first six but not in any particular order. With Internet Explorer 8, the first six requests that came back were not of the first six.

Note in the screenshot, we don't see the 2nd request coming back yet. If there's a

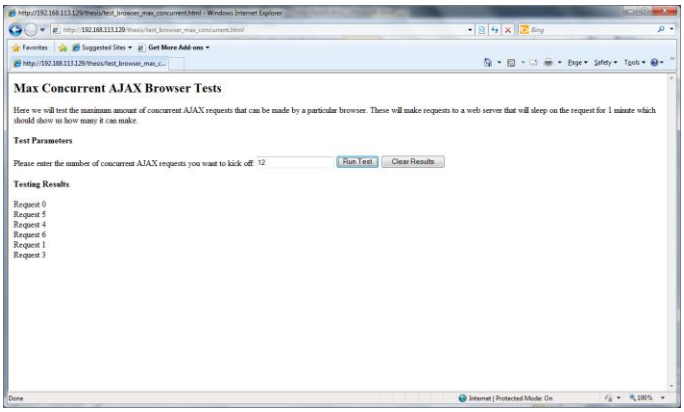

Figure 8

large enough difference between browsers, then this may affect how we partition if the number is greater than 6, since we will have no order in the way the page is responded and the first set of requests that we intend to send may not be the first set of requests that come back. Why would this matter? Well if we wanted to design it such that a large amount of the page as far as size is concerned is loaded first to give the user the experience that the page is loading in a reasonable amount of time, by having sets of requests, that is requests greater than 6, we would have no control potentially which one is being processed first by the web server.

Google Chrome 5.0

Running the test on Google Chrome, the same results occurred, only 6 concurrent connections, however we had a much more

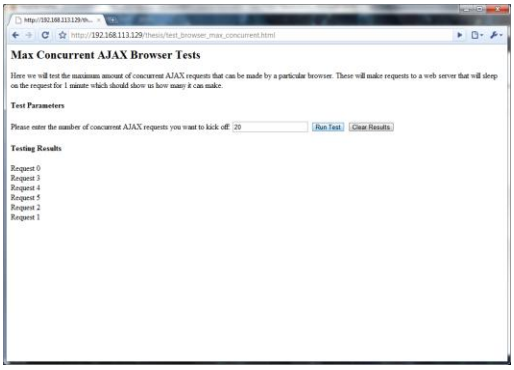

Figure 9

ordered set that came back, like that of firefox, meaning the first set of requests that came back were those that were of the first 6 requests.

Apple Safari 4.0

Apple Safari had the same results, only 6 concurrent connections. However, Safari's testing results showed that the order in which the requests were sent were much more random than that of other browsers.

If a user was using Safari to browse pages that are using the framework that we are going to create, and we have more than 6 concurrent requests being sent to the web server, then this could pose a serious problem if we are to want some sort of control over the order in which those requests are sent.

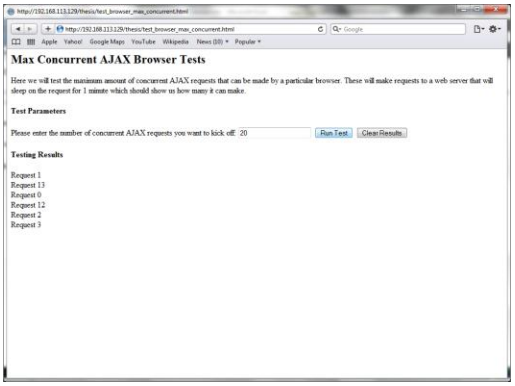

Figure 10

Apple Safari 5.0

During the beginning of this research, Apple Safari 5.0 came out and again had the same results as Apple Safari 4.0.

Verification

The first test for verification was to ensure that this restriction wasn't coming from the apache web server. We looked at this first because of the same number of concurrent requests limitation from all four major browsers. The first thing we checked was to make sure there were enough processes running:

Looking at the number of processes running on Apache, we noticed there were more than 6, so there were plenty of processes to handle more than 6 requests:

```
ps -ef | grep apache
root      /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
```

```
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
www-data  /usr/sbin/apache2 -k start
```

To also verify apache further, we checked the config which had the following:

```
<IfModule mpm_prefork_module>
    StartServers          5
    MinSpareServers       5
    MaxSpareServers      10
    MaxClients          150
    MaxRequestsPerChild   0
</IfModule>
```

What one would notice is that there are 5 StartServers, although our process listing showed that there were 10 which is the value for the MaxSpareServers. To make sure that this wasn't the issue, we modified the StartServers and MinSpareSErvers to a value much greater than 5 and the MaxSpareServers as well to 30, and retested:

```
<IfModule mpm_prefork_module>
    StartServers          20
    MinSpareServers       20
    MaxSpareServers     30
    MaxClients          150
    MaxRequestsPerChild   0
</IfModule>
```

After retesting, we noticed that all four major browsers still had 6 as the limit, looking online we were able to find official documentation for Internet Explorer 8 that this was indeed the case (8), however for Safari and Chrome, we could not find official documentation on those browsers, just forum and blog posts (15).

In Firefox, we were able to verify in the setting network.http.max-persistent-connections-per-server. Although we can change this value per browser, the goal of our implementation is such that existing browser settings are utilized. From this research, it

may prompt browsers to change this setting, but from various forum posts online, it appears that this setting is in place to both limit the load a user puts on a web server and for the stability of the web browser.

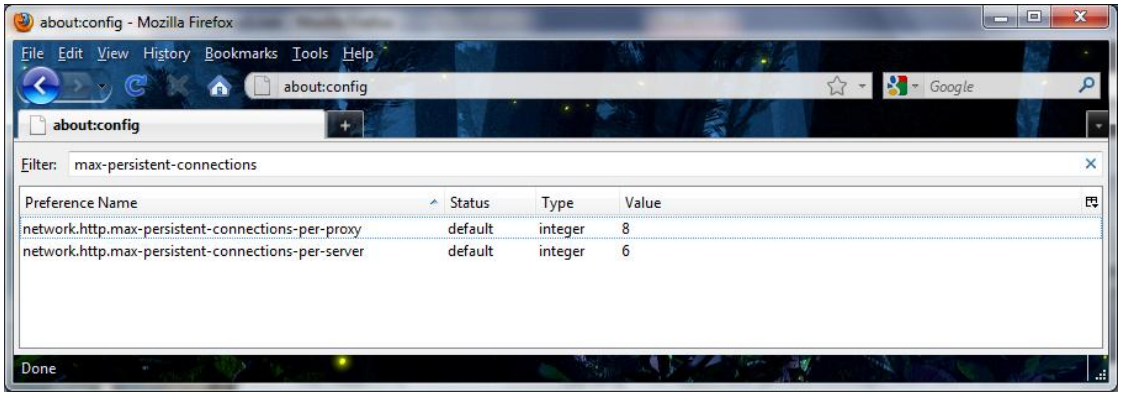

Figure 11

Conclusion of Testing Concurrent Connections

All four of the major browsers showed the restriction of 6 concurrent connections at a time to a web server. This has an obvious effect on the number of times we can partition a page to take advantage of parallel processing but this may also have an effect on the framework to be designed if the initial response we are sending back to the web browser does not have control of the partitions to be requested back.

This may make our partitioning scheme a little more difficult, because we will have to evenly distribute long requests against groups of 6, so that all long requests aren't in a group of 6 such that no requests process, this way other requests that are queued up still go through, however because of the randomness in the order the requests are sent, we have no control over this using AJAX unless we implemented a wait and release request mechanism.

CHAPTER V
TESTING ORDER OF CONCURRENT BROWSER REQUESTS

Testing Method

We tested the order of concurrent browser requests by recording in the sleep.php script to a file the order in which requests were made based on their ID that was created. For each test we ran ten sets, which is 60 requests since we can have 6 requests a set, and ran it 5 times.

The data would look like the following where each number is the request ID, the higher the ID, the later chronologically it was created:

```
1,2,4,5,6,7,9,8 …
3,2,1,5,4,6,7,8 …
4,1,2,3,5,8,6,7 …
```

We then took this data for the four major browsers, and created a score based on the following:

Take the average between the requests IDs received for that particular browser test, round that number to the nearest integer, subtract from it the index, and

take the absolute value from the result. Add all results together for each position to generate a difference score for that particular browser.

If we were to run this on the example data above for a particular browser, we would take the average of (1, 3, 4) = 2.667, round that number = 3, subtract the current position from it which in this case is 0, and take the absolute value which is 3.

Testing Results

After performing the tests, we noticed several behaviors. Before looking at these behaviors, here is a small sample from our testing data which looks like the following:

Slot Position	**0**	**1**	**2**	**3**	**4**	**5**
	Internet Explorer					
	1	0	7	6	8	9
	0	6	7	8	1	9
	0	6	7	9	8	10
	0	6	7	8	9	10
	1	3	2	4	5	0
Avg Diff from Position	0	3	4	4	2	3

Table II

Where “Avg Diff from Position” is the difference score from that current “Slot Position” or index. This gives us an indication of how unordered the requests are made to the web server from that particular browser. The numbers from the tests show that Firefox maintained the most order, and Safari maintained the least:

Browser	Difference Score
Internet Explorer	163
Mozilla Firefox	3
Apple Safari	267
Google Chrome	15

Table III

The scoring method is just an indicator of what type of order the requests are in based on how different the current request is from the index.

Considerations

Because of this randomness in major browsers, there is no guarantee that the order in which the client requests are made are the order in which they are sent to the browser. If we needed to have ordering in the requests sent to the browsers, we could check if the browser is “compatible” with that feature, meaning checking the User-Agent parameter from the head of the initial HTTP request sent from the browser to see if it is a browser that has a less randomness in the order requests are sent, in our testing it would only be Mozilla Firefox and Google Chrome.

CHAPTER VI
MANUAL PARTITION OF AN EXAMPLE PAGE

To get an idea on the performance gains of performing the dynamic partitioning and future design considerations, we created a sample page that contained several candidate partitions using the tag. We put a nested tag in there as well as we expect we will come across nested partitions to see what would be the best approach of handling them. Now in design of the framework, we are not restricted to tags, but will use them as an example as they are the predominant container tag in newer CSS design.

Approach

Looking at a sample of the code, we see some standalone tag as well as some nested tags where we outlined those areas:

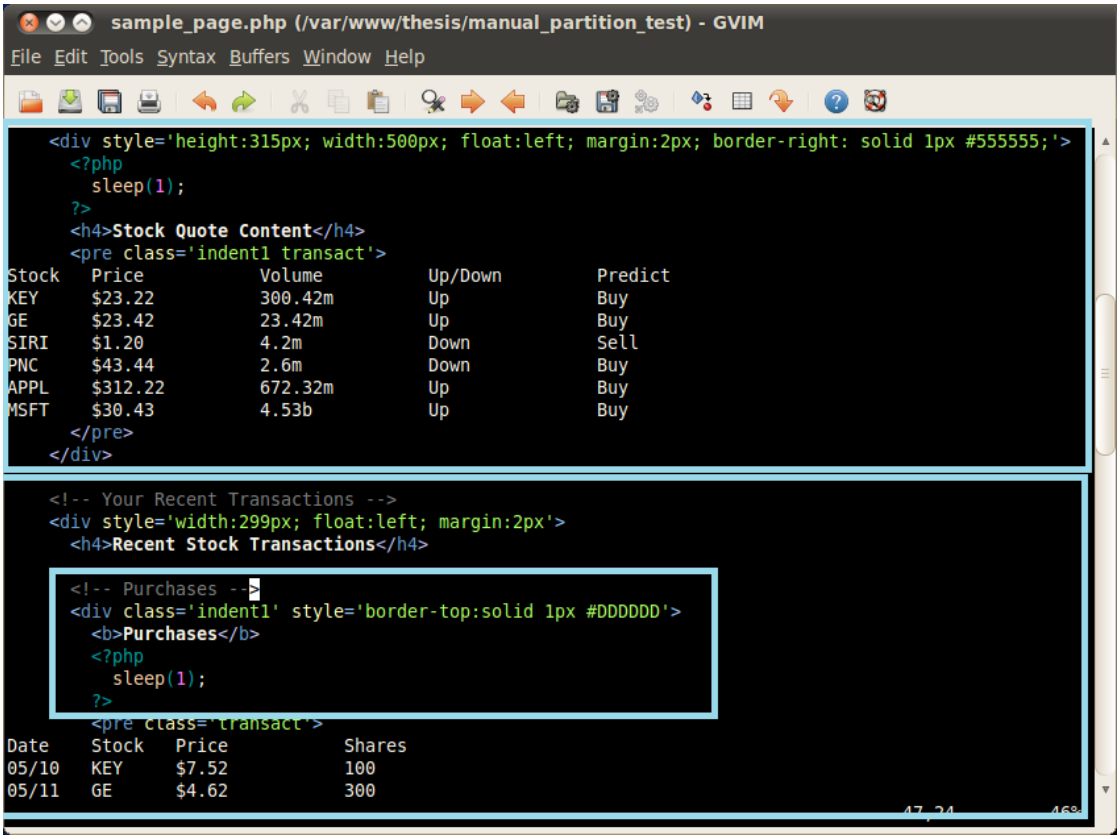

Figure 12

Which after rendering produces the following site where we again outlined the different partitions:

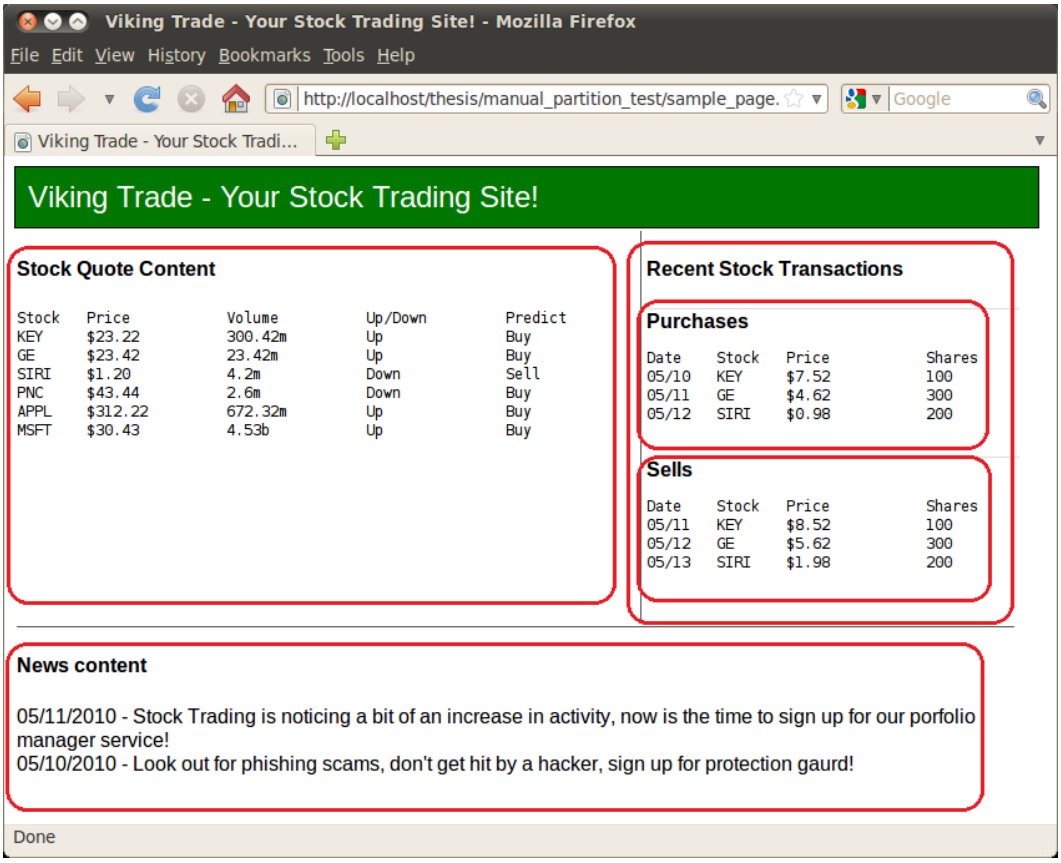

Figure 13

To do the manual partition, in creating the partitioned content so that it stands alone, there are two approaches we can do.

Separate File Approach

One approach is to separate the content of that partition, and store it in a separate file where the browser would make a request directly to that file. We would use the id attribute of the tag as part of the name of the separated content, if no ID existed, we would create one and store it in the tag:

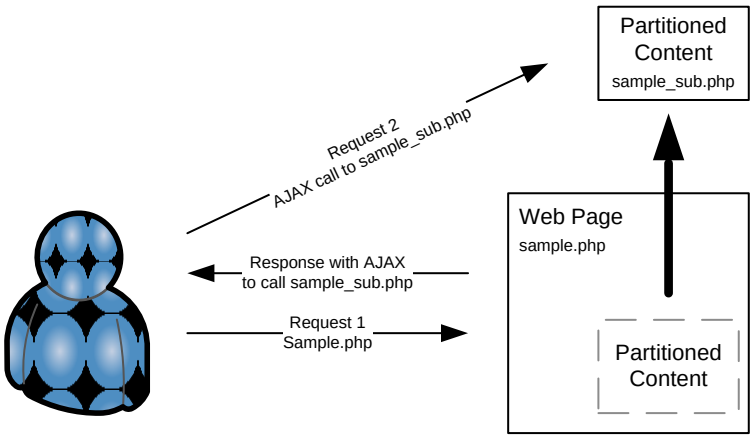

Figure 14

From the above diagram, the framework would separate the content and store it in a separate file. The sample.php page would then include AJAX to call the partitioned content, so that the initial request to sample.php returns the AJAX code to request the partitioned content, and the AJAX code would then place the response in the partitioned content area that it originated from.

Separate Method Approach

Another approach is to separate the content of that partition within the code from being executed by storing it in its own method, and having the browser as part of the AJAX code request for that method to be executed in that particular page, and the results returned to the browser will be placed where the partitioned content was removed:

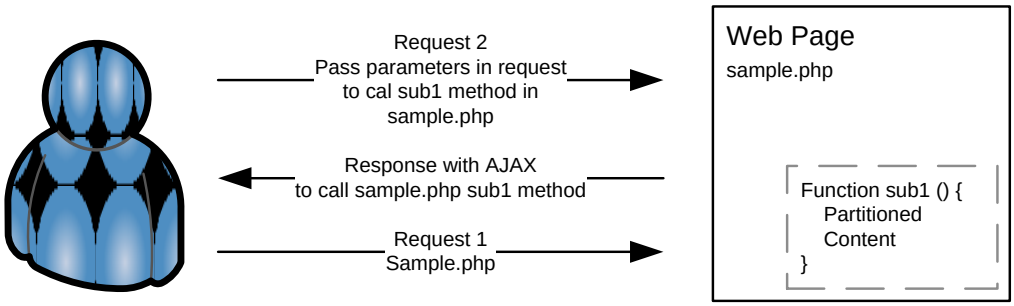

Figure 15

Just like in the Separate File approach, we can use the ID of the tag that existed or the one we generated to name the function. Our research will focus on the separate file approach.

Parsing the Page

In either approach, when we parse the page, we need to keep track of the partition structure. To do this, we will create a basic tree, with a parent/child relationship to represent the nested tag structure. When parsing the page if we perform dynamic partitioning at the child and at the parent, we need to partition the child first so that the AJAX code is created for the child, otherwise, when we take the partition of the parent out, it will include the child, and the code for the child will never be created.

Therefore as we walk our tree where each node represents a partition, we will need to check if there is a child, and if so go to the left-most child, and repeat. If there is no child, create the partition, move up to the parent, and delete the child where the

partition was created. We will repeat this until there are no more elements in the tree except the root which would be the <html> tag.

An example of how this tree would look includes the following based on our example page:

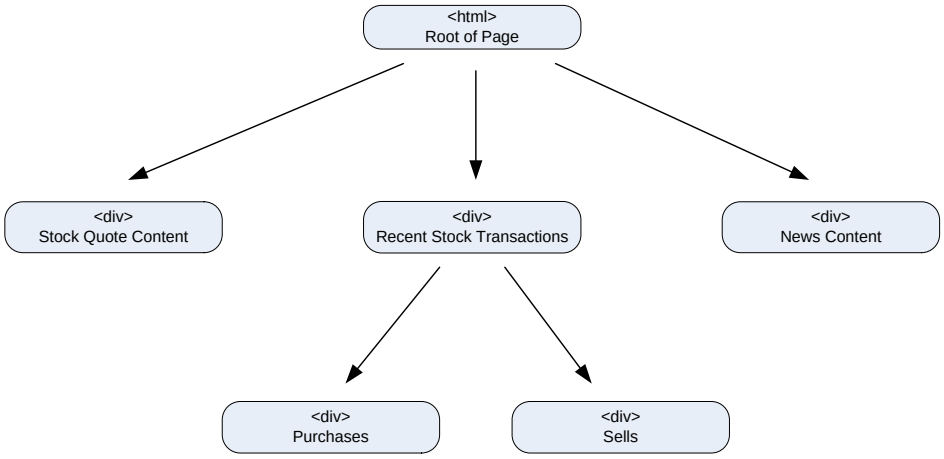

Figure 16

So walking through this tree, we would start at the root, go to the Stock Quote Content, there are no children, so create the partition, and then remove that element from the tree, then go to the Recent Stock Transactions node, then Purchases, there are no children, so write out the partition, and remove the purchases node, at this state, this is how our tree would look:

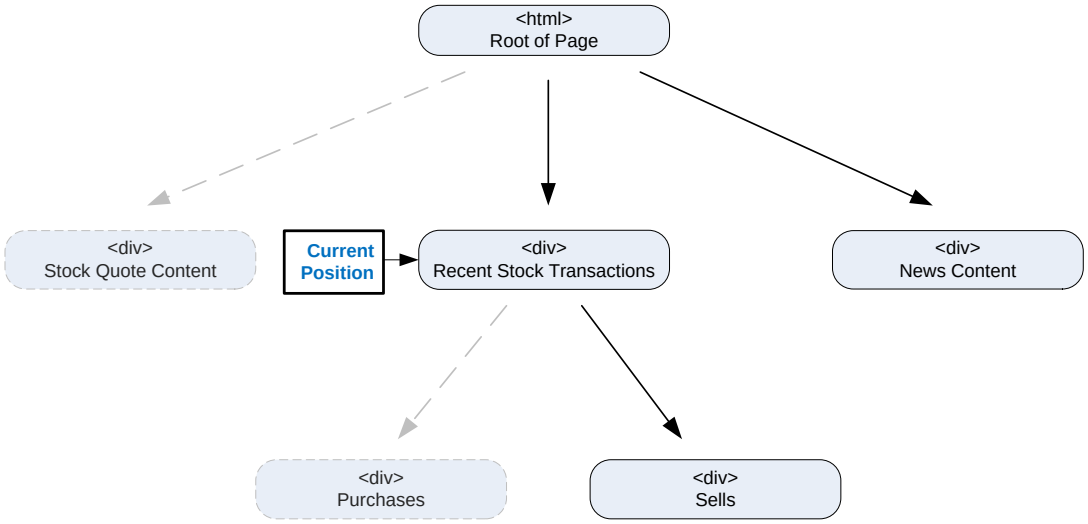

Figure 17

Once we remove all nodes from the tree with exception to the root, we are done. In our example, when we assigned IDs to the tags, we had the following mapping:

ID	Content
sub1	Stock Quote Content
sub2	Purchases
sub3	Sells
sub4	Recent Stock Transactions
sub5	News Content

Table IV

Where performing the Separate File approach, we had the following files created: result_page.php, result_page_sub1.php, result_page_sub2.php and so on.

Testing

The first part of the manual partition, we loaded the concurrent AJAX request library and created an array that would hold the concurrent AJAX request objects:

```
<!-- Insert reference to concurrentAjax.js library and build base code which will be the queue of requests-->
<script src='../common/js/concurrentAjax.js' language="JavaScript"></script>
<script>
  var cAjaxRequestQueue = new Array();
</script>
```

Figure 18

Following this, the next step was to build a tree of the content we wanted to divide. Since there are only 5 partitions to be created, it's not trivial to perform without a formal data structure. When we build the framework, we will construct a tree like data structure.

The first step was to remove the code from that partition, place it in its own file, and insert the AJAX code to request that content. In the screenshot below, we create a new cAjaxRequest object passing it the page we are requesting where it is the removed content, then the function to call once we get a response back which just takes the response and places it in the container where it was originally removed.

```
<!-- Stock Quote Content -->
<div id='sub1' style='height:315px; width:500px; float:left; margin:2px; border-right: solid 1px #555555;'>
  <!-- Insert code for concurrent AJAX request -->
  <script>
    // Create a new request adding one to the array by getting the current size and add 1, request the sub page we created, and
    // store the results in the <div> tag where we stripped the content from and stored in a sub page
    cAjaxRequestQueue[cAjaxRequestQueue.length] = new cAjaxRequest("result_page_sub1.php",
      function(response) {
        document.getElementById('sub1').innerHTML = response;
      }
    );
    cAjaxRequestQueue[cAjaxRequestQueue.length - 1].doGet();
  </script>
</div>
```

Figure 19

Before going further, we needed to test to make sure that this first step did indeed work. One thing we found was a bug in the randomness we added to the concurrent AJAX request to prevent caching. We would just append an & to the request with a random identifier, however since we don't have any URL parameters being added to the GET request, the & became part of the filename and we were getting HTTP 404 errors:

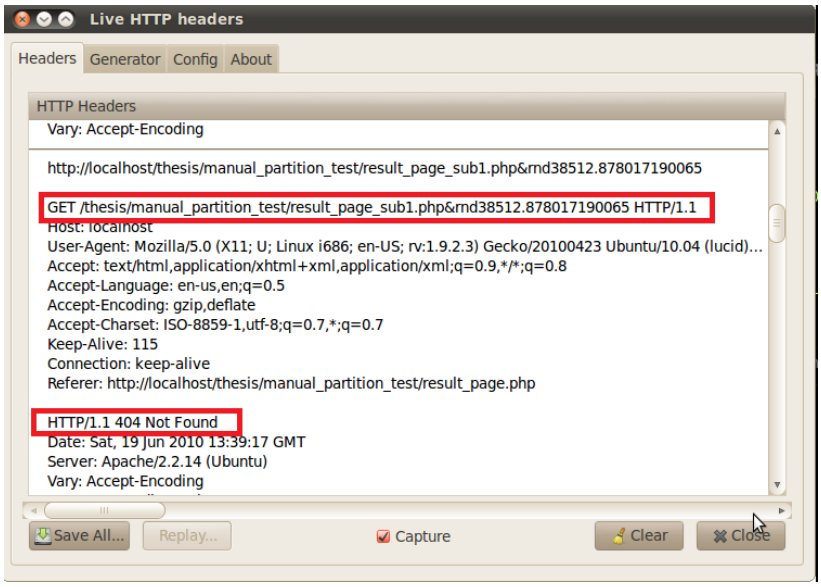

Figure 20

To get around this, we inserted code in the doGet method of the cAjaxRequest to check if we are passing any URL parameters in the request, and if so add the random string using an ampersand, otherwise if there are no URL parameters, let the random string be the first one by adding a ?:

```
// Function if we want to do a get
this.doGet = function() {
  // Need to check if there is a ? already in the URL to show a request, if there is, use an &, otherwise use a ?
  var addToUrl;
  // Check if ? doesn't exist
  if (url.indexOf("?") == -1) {
    addToUrl = "?";
  } else {
    addToUrl = "&";
  }

  // Add some randomness to the URL to prevent caching
  url = url + addToUrl + "rnd" + Math.random() * 50000;
```

Figure 21

After resolving this issue it worked, so we repeated the process for remaining partitions. On observation, the page that was manually partitioned loaded much faster.

Results

We load tested the manual partition with each browser and then load tested the pre-partitioned page with just one browser since the response times were close enough all browsers, and gathered the average response time, minimum response time, and maximum response time in milliseconds. The results from our testing showed a definite increase in performance using the partitioned approach where we saw almost a 4x increase in performance on a page with 5 partitions:

	Response Time		
Browser	**Avg**	**Min**	**Max**
Apple Safari	1093	1062	1147
Google Chrome	1108	1040	1192
Firefox	1180	1148	1345
Internet Explorer	1433	1388	2558
Firefox (Before Partition)	4172	4941	4086

Table V

Considerations

Some things we need to consider when building the framework for doing the dynamic partitioning are when separating to a file, is authorization that was built in the app being done, does code have access to local variables and libraries that it needs, with separation from method does it eliminate some of the complexities, also what about scope of variables?

CHAPTER VII
LOAD TESTING (WHERE IS THE BOTTLENECK?)

To help identify where the potential bottlenecks are with our partitioning approach, we evaluated several different performance monitors for Linux and found that “collectd” was the best one to use since it was highly configurable in the information that one would want to collect and have the ability to change the graphical view by zooming in and out and creating subgraphs. The goal of the testing we are doing here is to find the bottleneck on the system where the web server resides, is it the network, memory, processor, or even disk where we find bottlenecks in serving up the requests to the client, those questions we want to answer.

Testing Approach

Our testing approach was consistent throughout the two types of tests we did, one in which we test the page before partitioning and one in which we test the page after partitioning. In each test, we started a new browser every two minutes that would run in an infinite loop making requests to the web server. We then collected graphical results from kcollectd and from a custom script we wrote to gather memory utilization from the Apache2 processes and the collected processes. We wrote the script after

doing our initial testing to make sure that the increase in memory that we initially saw was indeed due to apache and not our monitoring agent.

Pre-Partitioned Load Test

Schedule

For the pre-partitioned load test, here was the schedule that we ran:

Time	Action
12:56	Started IE
12:58	Started Firefox
1:00	Started Safari
1:02	Started Chrome
1:06	Stopped Testing

Table VI

CollectD Results

Below are graphical results that were gathered from running collectd and configuring the graph to represent particular areas of interest:

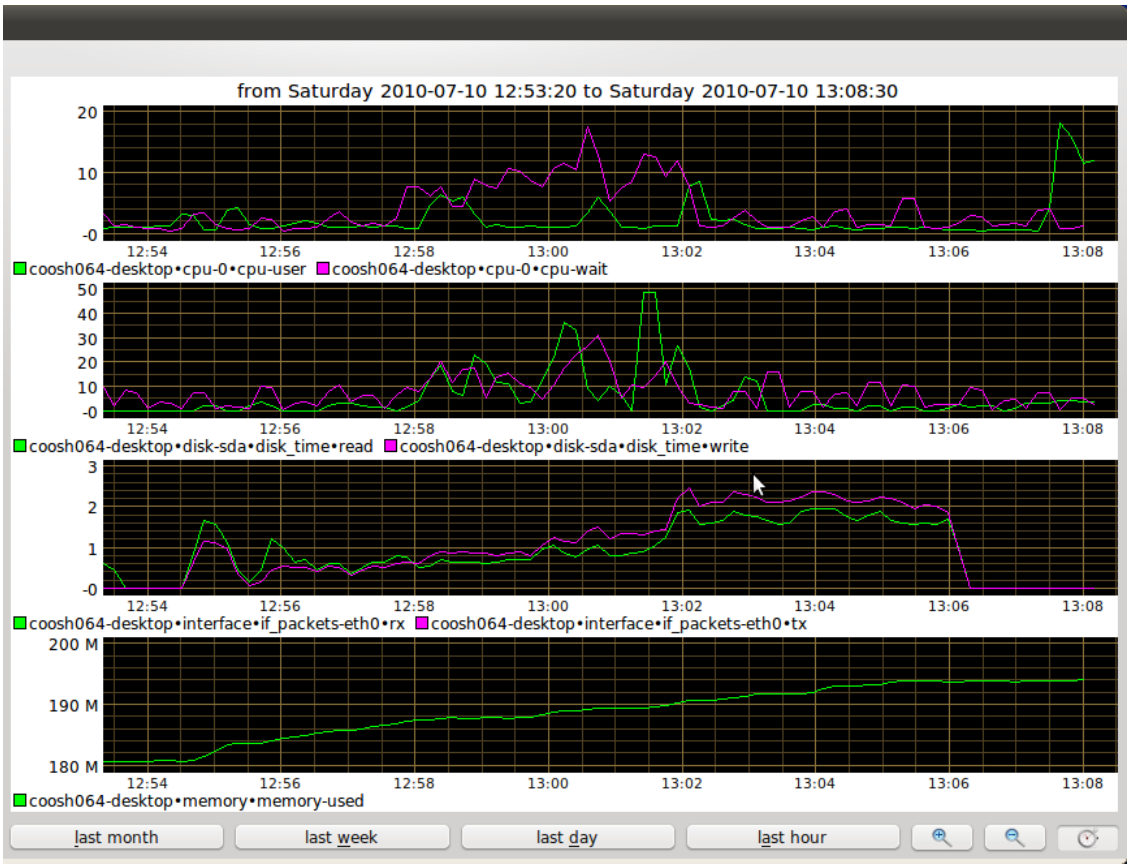

Figure 22

From this test the CPU both increased and decreased, which would appear to be fairly normal activity, and the same as the disk. The two pieces to show as potential bottlenecks that one can see a direct correlation with our testing is the network and memory utilization. At each interval of when a browser started testing, we noticed an increase of network activity, the same for memory. However with our network testing, once we started our last browser, we noticed a dramatic increase in the network utilization.

Memory Test Results

During the test, here is a summary of the results from the memory test script that we ran which ensured us that the increase in memory was due to the application and not our monitoring daemon, collectd. Also the memory increase was approximately 9% during the span of the test.

Before Testing

Current Memory Utilization by Apache : 18.6%, CollectD : 6.3% - 12:55:50

Current Memory Utilization by Apache : 19.4%, CollectD : 6.3% - 12:56:00

Started IE

Current Memory Utilization by Apache : 19.5%, CollectD : 6.3% - 12:56:10

Current Memory Utilization by Apache : 21.6%, CollectD : 6.3% - 12:57:52

Started Firefox

Current Memory Utilization by Apache : 22%, CollectD : 6.3% - 12:58:02

Current Memory Utilization by Apache : 22.7%, CollectD : 6.3% - 12:59:53

Started Safari

Current Memory Utilization by Apache : 23.3%, CollectD : 6.3% - 13:00:03

Current Memory Utilization by Apache : 24.9%, CollectD : 6.3% - 13:01:55

Started Chrome

Current Memory Utilization by Apache : 25.2%, CollectD : 6.3% - 13:02:05

Current Memory Utilization by Apache : 27.6%, CollectD : 6.3% - 13:06:08

Stopped Testing

Post-Partitioned Load Test

Schedule

For the post partitioned test, we ran a similar schedule but stopped the testing two minutes after the last browser was started:

Time	Action
20:57	Started IE
20:59	Started Firefox
21:01	Started Safari
21:03	Started Chrome
21:05	Stopped Testing

Table VII

CollectD Results

Below are the results from the statistics we gathered from collectd:

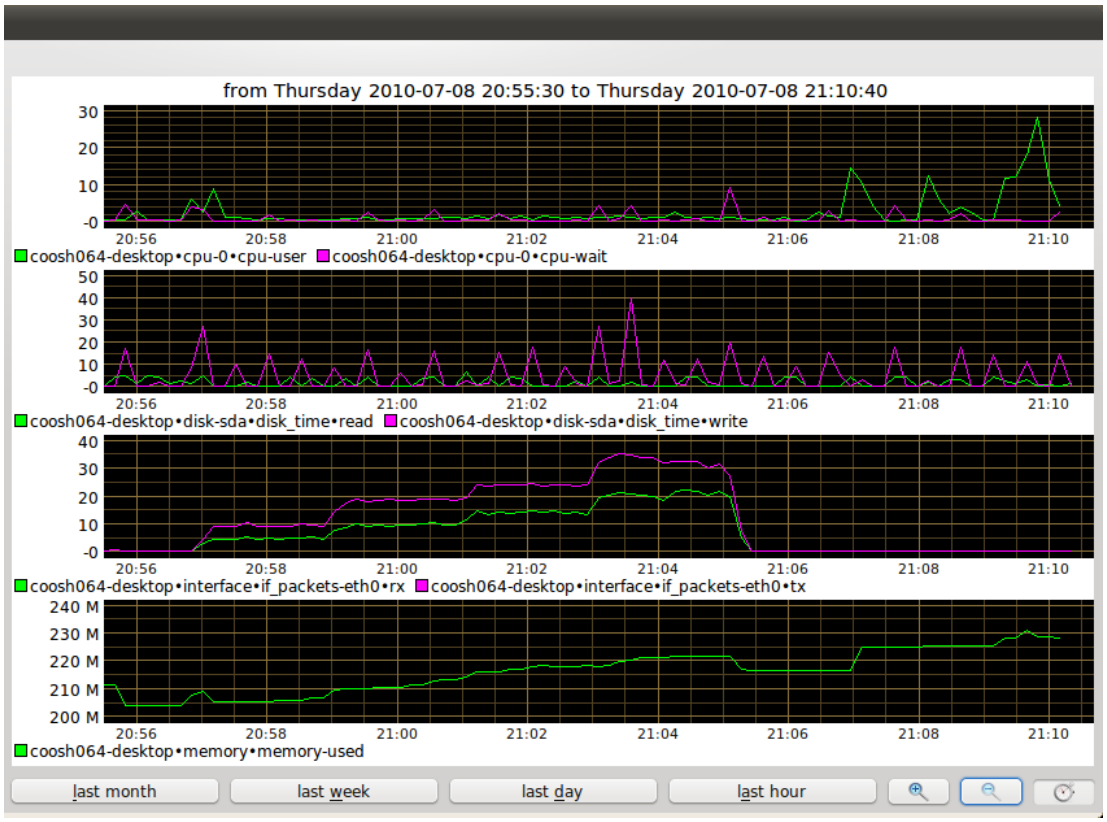

Figure 23

From this we can see again similar behavior for the CPU, disk, and memory, but where this test differentiates is that the network utilization had a much more dramatic increase when another browser started testing.

Memtest Results

From our memory test, we had the following summary of results, which showed that at the end of the test, the memory utilization increased by 28%.

Started IE

Current Memory Utilization by Apache : 13.2%, CollectD : 6.1% - 20:57:05

Current Memory Utilization by Apache : 18.4%, CollectD : 6.2% - 20:58:58

Started Firefox

Current Memory Utilization by Apache : 18.4%, CollectD : 6.2% - 20:59:08

Current Memory Utilization by Apache : 27%, CollectD : 6.2% - 21:00:59

Started Safari

Current Memory Utilization by Apache : 29.9%, CollectD : 6.2% - 21:01:09

Current Memory Utilization by Apache : 36%, CollectD : 6.2% - 21:02:51

Started Chrome

Current Memory Utilization by Apache : 38.4%, CollectD : 6.2% - 21:03:01

Current Memory Utilization by Apache : 41.2%, CollectD : 6.2% - 21:05:03

Ended Testing

Conclusion and Comparison of Testing

From the testing, we noticed more normal system behavior from the CPU and the disk. The two areas of interest that could be potential bottlenecks were the memory and the network.

Network

On the network side, the utilization was approximately 10 times more on the post-partitioned load test then the pre-partitioned load test. However, we are performing almost 4 times as many requests for the manual partitioned page then the pre-partitioned page, and we suspect the other difference is that since the data being retrieved from the web page is small, the HTTP header that is being sent along with the

request and in the response from each partition have a much greater proportion of the overall data in each request. So depending on what is in the partition will affect that proportion and potentially create a bottleneck in the network.

To test the performance difference within the network, we ran the load test with 20 requests a piece and a 5 second wait between each test, and had the following results:

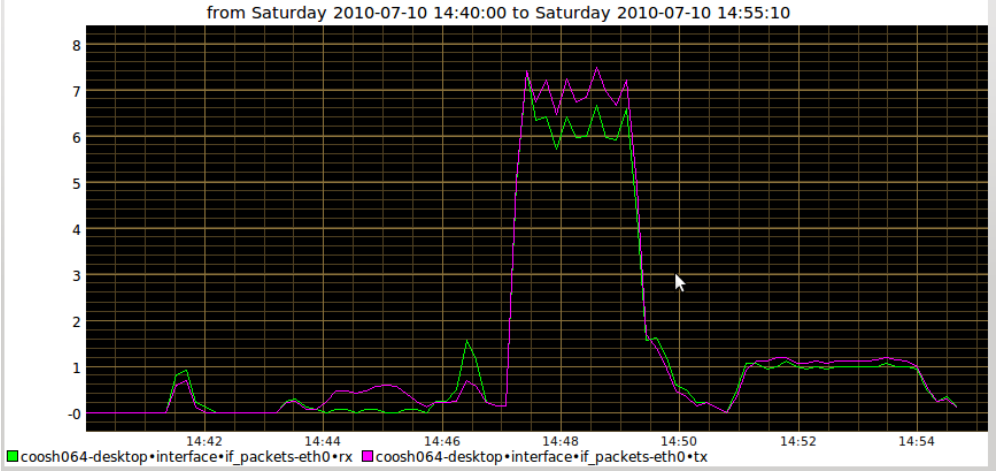

Figure 24

What we noticed here is that during the manual partition test, our network traffic was about 7x more but the test also completed in a smaller window, as opposed to the pre-partitioned page which took a little longer to complete. Part of the extra traffic is the HTTP header which responds with a smaller amount of data actually being sent back, so with the 5 partitions, we can expect a 7x more increase.

With our particular test, the size of the HTTP header for the request and the response is 792 bytes, multiply that by 4 for the partitions and we have a total of 3,168 bytes in HTTP header data. The actual data from the response in these partitions are a

total of, and the actual data is 852 bytes which is almost 4x as much data in the extra amount of HTTP header data as the number of actual data bytes.

With partitions that have more content, we can expect this difference to decrease but still exist. We will perform more testing at the end of this paper once we have a framework and we can divide several pages up and have multiple test scenarios.

Memory

The increase in memory wasn't as dramatic as the network. In percentage form it seemed to be, however when looking at the data from collectd, it was between 20MB to 30MB more, which is relatively small. If we were to rank our potential bottlenecks, our network is are greater concern and memory is a lesser concern.

Retest with Local Load Method

Previous Testing Approach

The initial load test was performed using sleep statements in each partition that would have the executing code sleep for one second before returning the remaining content. The sleep method was implemented with the idea that in a more enterprise environment for a distributed web application different components for the web application including a database, web service, directory, and other resources may reside on other systems and that calls to those systems would be idle time from the source system, the web server.

We implemented a method (17) which does both disk I/O and CPU processing based on some randomization in place of what we used before. Our implementation follows the following pseudocode:

```
function simulateLoad
  Loop through 2 times to hopefully do CPU and disk I/O in
  one call to simulateLoad
    Get random value p which equals  0 or 1
    if p = 0
      perform nested loop division of variables with
      random number of iterations
      between 200 and 350 times in outer loop
    if p = 1
      open file in /tmp and perform random writ eto file
      between 75,125 records
  End Loop
end function
```

Test Results

We re-ran the load/performance testing for the non-partitioned and partitioned web page that the original tests were ran against where we made 50 requests using each method and gathered the response times.

The first test was ran on a 1 core machine where we used VMware and specified 1 core which we verified by reading the /cpu/procinfo file. The results were the same with the partitioned page as the non-partitioned page with having the following averaged run times in milliseconds:

No Partition (50 Runs) 8979

Partition (50 Runs) 8708

The difference is very small and considering we have a randomness implemented in the load simulation function we can't definitively say that one performed better than another with the minimal difference in performance. We then changed the number of visible cores to 2 and reran the load test and the results from the response times showed that the response time was almost cut in half. Following are the average response times in milliseconds:

No Partition (50 Runs) 6475

Partition (50 Runs) 3647

From these results, we can see that the extra core increased the performance of the delivery of the web page.

Conclusion of Testing Results

The one issue with this testing approach however is that all processing is done locally on one machine. In a more enterprise environment, the partitioning method may be more advantageous for more distributed systems that have a database on a separate server, a web service on a separate server, etc where it utilizes that wait time that the web server is using to receive a response back from other systems to process other parts of the page. However, even when all resources are local, the partitioning method does utilize the server more by using multiple cores/processors to process requests. In this scenario, the CPU was the bottleneck as it spiked to 100% during the test.

CHAPTER VIII
DYNAMIC PARTITIONING

Designing the Parser

When looking at ways to do the dynamic partitioning, there were several approaches that we could take. One approach was to use a DOM parser that is available in PHP. We tested this approach first and found through our testing that the DOM parsers that are available are more suitable for traditional XML documents and not the kind of input that we would be working with where we will also have a mix of server side code and HTML. (13)

Designing our own parser, we would use regular expressions and build our own tree data structure to represent the nesting of elements and content. This will allow us to easily walk the tree and extract elements for the dynamic partitioning.

Our parser will function as follows:

1. Create a ROOT element in the tree
2. Extract Content (optional), div tag then Remaining Content
3. Create Content as child of current element
4. If we hit end tag, go back to #2

If we were to parse the following HTML document:

```
<html>
  <body>
    Welcome
    <div id='msg'>
      Content before nested div
      <div id='nested'>
        Nested Content
      </div>
      Content after nested div
    </div>
    Goodbye
  </body>
</html>
```

We would get the following tree data structure:

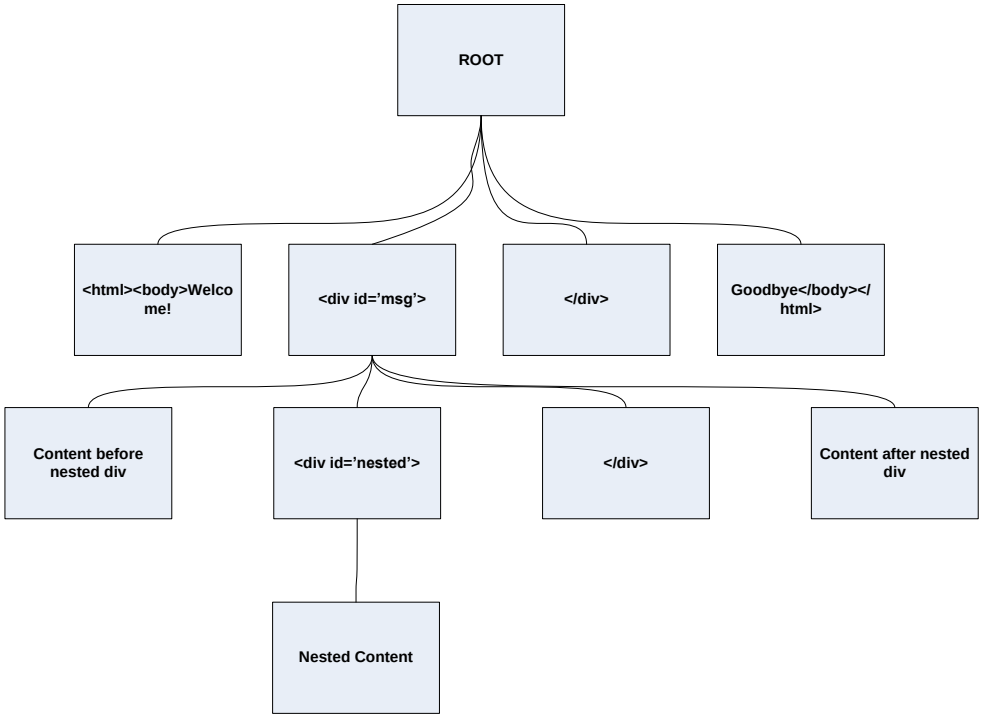

Figure 25

Once we have our tree data structure, we can then print out our HTML file by going to the left child that has not been accessed, printing its contents out, and repeating that process for each child that has not been accessed.

Implementation of Node Tree Structure in PHP

We built this implementation in PHP using an object oriented approach where we have a tree node object that can contain an array of children objects. These children objects would be other tree node objects. Other properties of this node contain an ID which would be used as the ID attribute in the div HTML tag, the tree node type which can be a nondiv, opendiv, and closediv, and the content of the node. Using the content of the node, if we walked the tree from the root element to the left most element and repeat this for each untouched node, we would print out all the content in order.

The tree walk method that we designed allows us to pass a callback method that will be ran on each node that the tree walk method reaches. This allows us to perform several operations on the tree with the same tree walk method.

ID Assignment

When making the concurrent AJAX requests, we need a unique ID for each partition. We designed the parser to use an existing ID if it exists, and if not, create a dynamic ID and increment it by one for each succeeding partition without an existing ID. This ID is then stored in the tree for quick retrieval as a property of the TreeNode class.

Separate File Approach

For this research, we implemented the separate file approach. To implement this approach, we had to come up with a way of storing the files effectively on the local filesystem. To do this, we create a directory where the parsed page is contained with a naming format of:

```
_<source_page>-dynpart
```

Within this directory, we store files based on the ID attribute of the Tree node. While we create these files however, we will more than likely have nested div tags:

```
<div id='1'>
  Content Before
  <div> Content Nested</div>
  Content After
</div>
```

In this scenario, we need two files for the content of the div tag with the ID of 1. One file will have "Content Before" as its content, the other will have "Content After". To work around this, we add a sub index to the file name. Following this approach, a div tag that has an existing ID would have the following file convention:

```
<id>_<sub_index>
```

And a dynamic generated ID would have the following file convention:

```
dynamic_partition_<dynamic id>_<sub_index>
```

Pseudo code of Parser

The parser was created in PHP and used regular expressions within the code to grab tokens which were defined as content before tags, tags, content within

 tags, and content after tags and stored them in the tree such. The core pseudo code for the parser is as follows, note that comments start with the #:

```
# Create partition tree from input file
Create root element for partition tree and set as current node
While file has content
      If remaining content has a div tag, grab content up to div
      tag and div tag
            Add content before div to tree as child of current
            node
            If div tag is open div
                  Add tag as child of current node
                  Set current node to just created child
            If div tag is close div
                  Add as child node to parent of current node
                  Set current node equal to parent
            Set remaining content equal to content after div tag
      Else
            Add content of remaining file content as child to
            current node
Return tree to parser

# Walk tree and add unique identifier for each div tag
Set current node equal to root node
function walkTree
      If current node is an open div tag
            If current node doesn't have ID attribute
                  Assign dynamic ID to node
      If current node has children
            Foreach child
                  walkTree of child

Prepare for dynamic partitioning by creating filesystem for
separate file method using input file name

# Dynamically partition the tree
function dynPartTree
      Foreach child of current node
            dynPartTree child
            If child type is within a div tag and is a nondiv
            type
                  Write child content to filesystem using ID
                  if concurrent AJAX library has not been
                  included
                        Include concurrent AJAX library in child
                        content
                  Set content of child = concurrent AJAX request
                  for child content on filesystem

# Walk tree and print out partitioned file to original file
Set current node equal to root node
function walkTree
      Write to file node content
      If current node has children
            Foreach child
                  walkTree child
```

The actual code for this parser can be found in Appendix A.

Execution of Parser

The execution of the parser successfully performed dynamic partitioning of the page in a similar structure of the manual partitioned page, thus yielding the same performance results as the manual partition.

CHAPTER IX
CONCLUSION

The research showed that there are definite performance gains to be achieved by performing partitioning of web pages using concurrent AJAX techniques. These performance gains can be as high as decreasing the amount of web page load time to the time it takes to load the largest partition. It also showed that it is possible to perform dynamic partitioning of the web pages so that less developer involvement is required.

Optimal Scenarios

The optimal scenario to use partitioning of a web page is when most processing of the web page resides on separate systems such as database servers, web services, legacy systems, etc. This optimal scenario allows the web server to do minimal processing and work on other requests while it waits for results from separate systems.

Another scenario is where one would have processing all local to the web server. In this scenario, performance gains can still be achieved. As long as the web server contains more than one processing core, two partitions can be processed at one time, therefore dividing the processing time by the number of cores.

The worst case scenario is where all processing is local to the web server and there is only one processing core. In this scenario, there will be no performance gains, however the processing time may take longer since there is more overhead in making additional requests.

When to Use Partitioning

The partitioning has showed to increase the load on the web server. This makes sense since we are requesting the web server to process more requests at once and sending these requests concurrently instead of sequentially in the form of AJAX. When using this partitioning technique, the processing systems should be scaled appropriately to handle the requests. In a small user base, this is mostly not a concern, however with a larger user base where the web architecture of the application will see more requests planning and testing has to be done to make sure the architecture can handle the extra load. The benefit of course is a greater user experience and faster web page delivery times.

Further Research in Dynamic Parser

Further research can be done in designing the dynamic parser. In most cases to make improvements, the parser would have to be able to recognize the source code at a compiler level, requiring a more significant amount of work. Such issues such as relative addressing in partitioned content could be resolved, as well as required code that would need to be in each partitioned content. One way to circumvent this complexity is to have custom tags with an XML like structure that the developer would insert into their

page to identify common code to include in each page such as authentication/authorization and required libraries, as well as a custom tag to define each partition. This would also give more control of the developer to do user testing to create a better partition design of each page based on its needs and performance. Also by doing this approach using custom tags, we avoid any wasted partitions of div tags that contain very small content. If this approach was to be used, to resolve the relative addressing issue, the dynamic partitions could just be stored as hidden files locally on the web server and not in a separate directory. Another area would be to design the parser to partition static content in a format that could be cached locally by the browser since most parts of a dynamic page are by nature static content.

BIBLIOGRAPHY

1. AJAX Tutorial. 21 October 2010 <http://www.w3schools.com/ajax/default.asp>.

2. Cornford, Richard. Javascript Closures. 21 October 2010 <http://www.jibbering.com/faq/notes/closures/>.

3. Eckstein, Robert and Stephen Spainhour. Webmaster in a Nutshell. O'Reilly Media, 1999.

4. eroman@chromium.org. Issue 12066 - chromium - Match Firefox's per-host connection limit of 15 - Project Hosting on Google Code. 15 May 2009. 22 May 2010 <http://code.google.com/p/chromium/issues/detail?id=12066>.

5. Huang, Yin-Fu and Jhao-Min Hsu. "Mining web logs to improve hit ratios of prefetching and caching." Knowledge based Systems 21 (2008): 62-69.

6. JavaScript Array Object. 21 October 2010 <http://www.w3schools.com/js/js_obj_array.asp>.

7. Lerdof, Rasmus and Kevin Tatroe. Programming PHP. O'Reilly Media, 2002.

8. Microsoft. AJAX - Connectivity Enhancements in Internet Explorer 8. 21 October 2010 <http://msdn.microsoft.com/en-us/library/cc304129(VS.85).aspx>.

9. Niederst, Jennifer. Web Design in a Nuthsell. O'Reilly Media, 1998.

10. Peng, Tao, Changli Zhang and Wanli Zuo. "Tunneling enhanced by web page content block partition for focused crawling." Concurrency and Computation : Practice and Experience (2007): 61:74.

11. Philip. SpeedGuide.net :: Firefox / IE Browser Tweaks. 4 October 2005. 29 May 2010 <http://www.speedguide.net/read_articles.php?id=2448>.

12. PHP: DOM - Manual. 21 October 2010 <http://php.net/manual/en/book.dom.php>.

13. Pons, Alexander P. "Improving the Performance of client Web object retrieval." The Journal of Systems and Software 74 (2005): 303-311.

14. Regular expression - Wikipedia, the free encyclopedia. 21 October 2010 <http://en.wikipedia.org/wiki/Regular_expression>.

15. Stack Overflow. How many concurrent AJAX (XmlHttpRequest) requests are allowed in popular browsers? 21 October 2010 <http://stackoverflow.com/questions/561046/how-many-concurrent-ajax-xmlhttprequest-requests-are-allowed-in-popular-browser>.

16. Wong, Clinton. Web Client Programming with Perl. O'Reilly Media, 1997.

17. Trivedi, K.S. Probability and Statistics with Reliability, Queueing, and Computer Science Applications. Prentice Hall, 1982.

APPENDIX A
DYNAMIC PARTITION PARSER PHP CODE

```
#!/usr/bin/php -f
<?php
  // First check if we want to get a help for usage
  if ($argc == 1 && $argv[1] == 'help') {
    echo "\nUsage: dynPartPage.php source_file\n\n";
    exit();
  }

  // Then check for the arguments passed to the user, if the number of
arguments equals the number of arguments
  // equals the number of arguments we need, don't prompt the user,
otherwise prompt the user for everything
  if ($argc == 2) {
    // Get the input file
    $input_file = trim($argv[1]);
  } else {
    // Prompt the user for a source file
    $input_file = getInput("Enter file to convert");
  }

  $output_file = $input_file . "_new";

  // Perform input validation
  if (!file_exists($input_file)) die("Error: File ($input_file) does
not exist\n");

  // Grab the suffix of the file
  preg_match("/.*?\.(.*)/", $input_file, $suffix);
  $suffix = $suffix[1];

  // Create a tree from a source html file
  $root = createTree($input_file);

  // Walk the tree, calling addIdentifier callback
  walkTree($root, 'addIdentifier');

  // Prep dynamic partititon creates the filesystem data structure
needed
  prepDynamicPartition($input_file);
```

```
  // This does the magic and dynamically partitions page
  dynamicPartitionTree($root);

  // Open the output file, and call walkTree with callback of
writeToFile which will print the node content to the file
  $fh = fopen($output_file, "w");
  fwrite($fh, walkTree($root, 'writeToFile'));
  fclose($fh);

  // Now that we made it this far, rename the partitioned file and move
the newly created one on this one
  $backup_file_name = $input_file . ".predynpart";
  $i=0;
  while (file_exists($backup_file_name)) {
    $backup_file_name = $backup_file_name . "_$i";
    $i++;
  }
  if (rename($input_file, $backup_file_name)) {
    if (! rename($output_file, $input_file)) {
      echo "Failed to move $output_file to $input_file, exiting\n";
    }
  }
  else {
    echo "Failed to move $input_file to $backup_file_name, exiting\n";
  }

  echo "Successfully created partition page!\n\tStored pre-partition
page at $backup_file_name\n\tCreated dynamic partition content in
$dir_name\n\n";

  function getInput($prompt) {
    echo $prompt . " : ";
    return trim(fgets(STDIN));
  }

  function writeToFile($node) {
    global $fh;
    fwrite($fh, $node->content);
  }

  function printContentCallback($node) {
    echo $node->content;
  }

  function prepDynamicPartition($file_name) {
    global $dir_name;
    $dir_name = "_" . $file_name . "-dynpart";
    if (is_dir($dir_name)) {
      $dh = opendir($dir_name);
      while (false != ($file = readdir($dh))) {
        unlink($dir_name . "/" . $file);
      }
      rmdir($dir_name);
    }
    mkdir($dir_name);
  }
```

```
  function dynamicPartitionTree($node) {
    global $dir_name;
    global $first_pass;
    global $suffix;
    $children = $node->getChildren();
    foreach($children as $child) {
      dynamicPartitionTree($child);
      if ($child->isindiv && $child->type == "nondiv") {
      // Create our id and filename
      $id = $child->parent->id . "_" . $child->parent->partition_count;
      $file_name = $dir_name . "/" . $id . "." . $suffix;

      // Open file handler, and write the content, and close the file
handler
      $fh = fopen($file_name, "w");
      fwrite($fh, $child->content);
      fclose($fh);

      // Check if we made our first pass, if we didn't, then add the
script content
      if ($first_pass != "done") {
        $child->content = "<script src='../common/js/concurrentAjax.js'
language='JavaScript'></script> " .
                      "<script> var cAjaxRequestQueue = new Array();
</script>";
        $first_pass = "done";
      }
      else {
        $child->content = "";
      }

      $child->content .= "
        <span id='" . $id. "'></span>
        <script>
          cAjaxRequestQueue[cAjaxRequestQueue.length] = new
cAjaxRequest('$file_name',
            function(response) {
            document.getElementById('$id').innerHTML += response;
            }
          );
          cAjaxRequestQueue[cAjaxRequestQueue.length - 1].doGet();
        </script>
      ";

      // Incremenet the parent partition count
      $child->parent->partition_count++;
      }
    }
  }

  // This will add a unique identifier to each div tag
  function addIdentifier($node) {
    // Check to see if we have an open div
    if ($node->type == "opendiv") {
      // If we do have an open div, extract the ID attribute, and store
it in the object
```

```
      $id_pattern = "/.*?id\s*?=[\'\"](.*?)[\'\"].*?[\s\>]/si";
      $nonid_pattern = "/(<div)(.*)/si";
      if (preg_match($id_pattern, $node->content, $matches))  $node->id
= $matches[1];
      // Else, add an ID
      else {
      preg_match($nonid_pattern, $node->content, $matches);
      $node->id = getUniqueId();
      $node->content = $matches[1] . " id='" . $node->id . "' " .
$matches[2];
      }
    }
  }

  // This will create a unique ID and return it
  function getUniqueId() {
    global $id;
    if (! isset($id)) $id = 10000;
    else $id++;
    return "dynamic_partition_$id";
  }

  // Function to walk tree in order the way the elements were added,
allows you to pass the callback function
  function walkTree($current_node, $callback) {
    // Call the callback on our current node
    $callback($current_node);

    // Check if our current node has a child, if so go through all of
them
    if ($current_node->getChildCount() > 0) {
      // Get list of children, and make a recursive call to walkTree
for each child
      $children = $current_node->getChildren();
      foreach($children as $child) walkTree($child, $callback);
    }
  }
  // Will need to have a separate node called closediv, that will close
a previous tag
  // Tree node types nondiv, opendiv, closediv
  class TreeNode {
    function TreeNode($type, $content, $parent) {
      $this->type = $type;
      $this->content = $content;
      $this->parent = $parent;
      $this->children = array();
      $this->id = "";
      $this->partition_count = 0;

      if ($this->parent->type == "opendiv" || $this->parent->isindiv ==
true) $this->isindiv=true;
      else $this->indiv=false;
    }
    function addChild($child) {
      array_push($this->children, $child);
    }
    function getChildCount() {
```

```
      return sizeof($this->children);
    }
    function getChildren() {
      return $this->children;
    }
  }

  // This function returns a Tree structure
  function createTree($source_file) {
    // Store the source file in a single string
    $source_file = file_get_contents($source_file);

    // Create root and store it in current_node
    $root = new TreeNode("root", "", "0");
    $current_node = &$root;

    // Keep going while the source file contents are > 0
    while(strlen($source_file) > 0) {
      // Check for any type of div tag, have the s at the end of the
reg ex to span multiple lines
      if (preg_match("/(.*?)(<\/*?div.*?>)(.*)/si", $source_file,
$matches)) {
      // Add nondiv element which is the content before the div
      $current_node->addChild(new TreeNode("nondiv", $matches[1],
$current_node));

      // Check if we have a beginning div, or an end div, first check
for an end div by checking for a / in the tag
      // First check if we have an end by checking if there is a / in
the tag
      if (preg_match("/.*?\/.*/si", $matches[2])) {
        // Add the close div to the parent of this child
        $current_node->parent->addChild(new TreeNode("closediv",
$matches[2], $current_node->parent));

        // Point the current node to the parent
        $current_node = $current_node->parent;
      }
      // Else we have an open tag, so sent that to the current node, so
we can place the children underneath it
      else {
        // Create a temporary node, and add it to the current node
        $temp_node = new TreeNode("opendiv", $matches[2],
$current_node);
        $current_node->addChild($temp_node);

        // Store in current node the node we just created since we will
now be adding whatever it contains to this
        $current_node = $temp_node;
      }

      // Store the remaining match into the source file
      $source_file = $matches[3];
      }
      // Else, if we don't have any divs left in the source, add to the
current node which should be the root the left over content
      else {
```

```
      $current_node->addChild(new TreeNode("nondiv2", $source_file,
$current_node));
      $source_file = "";
      }
    } // End of going through the source file

    // Return the root node so we can print out the tree
    return $root;
  } // End of createTree function
?>
```

APPENDIX B
CONCURRENT AJAX JAVASCRIPT CODE

```
/*
  cAjaxRequest is a model for doing concurrent Ajax Requests by using
closures
*/
function cAjaxRequest(url, callback) {
  // Get a new XMLHttpRequest object
  var req = init();

  // Function to call when status changes
  req.onreadystatechange = processRequest;

  // Get a new XMLHttpRequust object
  function init() {
    if (window.XMLHttpRequest) {
      return new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      return new ActiveXObject("Microsoft.XMLHTTP");
    }
  }

  // Every time the status changes, check if the request was completed
  function processRequest () {
    if (req.readyState == 4) { // Check for completion
      if (req.status == 200) { // Check for HTTP 200 return
        if (callback) callback(req.responseText); // Run the function
that was passed with callback, this is a closure, pass the response to
it
      }
    }
  }

  // Function if we want to do a get
  this.doGet = function() {
    // Need to check if there is a ? already in the URL to show a
request, if there is, use an &, otherwise use a ?
    var addToUrl;
    // Check if ? doesn't exist
    if (url.indexOf("?") == -1) {
      addToUrl = "?";
    } else {
```

```
      addToUrl = "&";
    }

    // Add some randomness to the URL to prevent caching
    url = url + addToUrl + "rnd" + Math.random() * 50000;
    req.open("GET", url, true);
    req.send(null);
  }

  // Function if we want to do a post
  this.doPost = function(body) {
    req.open("POST", url, true);
    req.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
    req.send(body);
  }
}
```