WikiBench: A distributed, Wikipedia based web application benchmark

Master thesis by Erik-Jan van Baaren
Student number 1278967
erikjan@gmail.com

Under the supervision of:
Guillaume Pierre
Guido Urdaneta

Vrije Univesiteit Amsterdam
Department of Computer Science

May 13, 2009

Abstract

Many different, novel approaches have been taken to improve throughput and scalability of distributed web application hosting systems and relational databases. Yet there are only a limited number of web application benchmarks available. We present the design and implementation of WikiBench, a distributed web application benchmarking tool based on Wikipedia. WikiBench is a trace based benchmark, able to create realistic workloads with thousands of requests per second to any system hosting the freely available Wikipedia data and software. We obtained completely anonymized, sampled access traces from the Wikimedia Foundation, and we created software to process these traces in order to reduce the intensity of its traffic while still maintaining the most important properties such as inter-arrival times and distribution of page popularity. This makes WikiBench usable for both small and large scale benchmarks. Initial benchmarks show a regular day of traffic with its ups and downs. By using median response times, we are able to show the effects of increasing traffic intensities on our system under test.

Contents

1 Introduction

Although originally a place designed for researchers to easily exchange information, the world wide web has become one of the most important information infrastructures of modern society. We have seen a rapid transition from static HTML documents to advanced web applications like online email, social networks and online office tools such as word processing and spreadsheets. While such web applications became more and more common over the past years, only a limited number of web application benchmarking tools emerged. Hosting advanced web applications can require lots of resources. It is therefore important to perform research on how to improve various aspects ([11], [12], [8], [16]) of such hosting systems. Web application benchmarks are especially important when doing such research, since they provide a configurable, reproducible and often realistic simulation of real web application usage. Benchmark tools aid the systematic research into the performance of web hosting systems and make it possible to compare different systems and different system setups.

There are a number of benchmark applications that are used today, like TPC-W, RUBBoS and RUBiS. These benchmarks have similar characteristics. TPC-W simulates a web store, while RUBBoS is a simple bulletin board system and RUBBiS mimics an online auction site. Although these tools have proven to be useful to many researchers, they have limits in terms of data set size, scalability and functionality. For example, all three tools run on a single system. There is no built-in way to scale up to multiple systems. Although it can be subject to heated debate, we feel that the synthetic workloads these benchmarks create are unrealistic and lack configurability. Another important downside of these benchmarks is that they generate a constant load through a fixed number of emulated browsers. These emulated browser all wait indefinitely for a server to answer a request, while in reality a visitor is only prepared to wait for a limited amount of time, like 4 to 8 seconds [7]. In addition, the number of visitors typically varies greatly depending on the time of day, while most benchmark tools show a constant load over the entire time period. So these tools lack realism, flexibility and configurability: qualities that we think are very important when it comes to the development and testing of advanced web hosting setups.

To address some of the shortcomings of the currently available benchmark tools, we created WikiBench. WikiBench has a number of advantages compared to the previously discussed tools. First of all, Wikibench offers a high degree of realism, since it is entirely based on the Wikipedia software and data. we have obtained access traces from the WikiMedia Foundation. These traces contain detailed traffic logs of requests made to Wikipedia by its

users. We are able to convert these access traces to benchmark workloads by using our specialized TraceBench tool. TraceBench can reduce the intensity of this traffic while maintaining important traffic properties, allowing us to create very realistic benchmark workloads with intensities ranging from very low up to the original traffic intensity of the trace file.

To match the server side software with the workload files, we use the open source MediaWiki application [1], the software used to run Wikipedia. This application is quite advanced and has been tested extensively. In addition we have used publicly available snapshots from the WikiMedia foundation to build a mirror of the English Wikipedia site. So we now have a real world web application with a large amount of data to serve.

Since Wikipedia has a large and constantly increasing amount of data and visitors, basing a benchmark tool on this data is not an easy task. We have designed WikiBench, from the ground up to be an inherently distributed application. It can scale up from one machine for small benchmarks to many machines working together in a coordinated fashion. All put together, we believe we have created a benchmarking tool that is able to create a workload which matches reality closely. By using real world server side software and data, we think the WikiBench benchmarking suite is a very realistic and flexible research tool.

Initial benchmark results show a typical day of Wikipedia traffic and the relation between the request rate and the server response times. The intensity of this traffic is reduced with our TraceBench tool to fit our system under test.

The rest of this thesis is organized as follows. In Section 2 we discuss a number of existing web application benchmarks, in Section 3 we describe the WikiBench design in detail. In Section 4 we focus on how we create realistic workloads of arbitrary size from the trace files. Section 5 discusses our initial results and section 6 concludes.

2 Related Work

One of the more well known and intensively used tools to benchmark application hosting systems is TPC Benchmark W^{TM}[13]. Another tool we will discuss, intended for benchmarking caching servers, is called Web Polygraph[10]. Although not aimed at application benchmarking, it does have a number of interesting features.

2.1 TPC-W

TPC Benchmark W^{TM} (TPC-W) is a transactional web benchmark[13]. The workload is performed in a controlled internet commerce environment that simulates the activities of a business oriented transactional web server. This is done by implementing a web store and a testing tool that browses through this store. TPC-W uses the concept of Emulated Browsers (EBs). Each EB runs in a separate thread and emulates a user's browser and the user actions. The EB uses Markov chains to find random paths in the TPC-W store. An EB has a random think time that emulates the time a user takes before clicking on the next link. This think time is distributed in such a way that the average think time is 7 seconds. Load generation in TPC-W happens in a best-effort fashion. This means that the EBs wait for each request to be processed, no matter how long this takes. The result being that this benchmark tool can only vary load by changing the degree of concurrent emulated browsers. instead of changing the request rate, which would match reality much closer.

TPC-W offers three different modes. These modes have different ratios between the amount of information requests (page reads) and the number of orders places by customers. In the default mode, the ratio of read and order pages is 95%/5%. In browse mode, users mostly browse around with 98% browsing and 2% ordering. In order mode, users mostly buy stuff and order pages are visited 50% of the time. Order mode puts much more pressure on the database system of a TPC-W site, since the actions associated with the ordering of products are not cacheable. So the latter mode can be used to put more pressure on the database without having to increase the web server capacity. TPC-W Benchmark has been discontinued since 2005, but is still used for testing distributed hosting systems.

Two other commonly used benchmark tools are called RUBBoS [2] and RUBiS [3]. Both have similar characteristics to TPC-W. RUBBoS is a bulletin board benchmark modeled after an online news forum like Slashdot. RUBiS is a benchmarking tool based on an online auction site like eBay. Both RUBBoS and RUBiS make use of emulated user sessions, transition

tables with probabilities and wait times.

Although web site traffic does often have returning and predictable patterns, we feel that the workloads of these tools are too limited. A larger problem we see is that the emulated browsers create a very stable and predictable workload on a system. These tools, for example, do not create spikes of traffic at the page level, while in reality pages get linked on large news forums like Digg or Slashdot. We call these traffic spikes flash crowds. A single page may, for short periods of time, easily receive many times more traffic than it gets on average. For a distributed hosting system it is easy to anticipate the stable traffic pattern and page popularity distribution that Markov chains create. But in reality a distributed hosting system will have to deal with more difficult traffic patterns like the page level flash crowds described above. Such flash crowds, even small ones, change the spatial and temporal locality of the requests, possibly overloading parts of the system if no timely measures are taken.

Another problem we see with the emulated browsers in the described benchmarking tools is that they use a fixed amount of concurrent emulated browsers to put load on the system under test. So the load put on a hosting system is defined by the amount of emulated browsers instead of by the request rate. If the server is overloaded, the emulated browsers will all wait longer, decreasing the load put on the server at some point. Workload and response time are therefore directly related to each other. This is however not representative of real world traffic, where new and current visitors will keep hammering a hosting system with new requests. In other words, these benchmarks define server load by the number of concurrent emulated browsers, instead of by the number of requests per time unit a server actually can handle before it starts to react unacceptably slow.

All three synthetic traffic generators also lack the traffic that is generated by web crawlers, malicious users and page scrapers. Such users can for example have a very constant think time and different page preferences than a regular visitor. A crawler will generally visit all pages in a site, while most regular users will only visit a small subset. A malicious user might submit loads of difficult search queries or extraordinary amounts of page edits to vandalize the content. Page scrapers might only download the page and not the related items, like images and style sheets. These users exist in the real world, so a system should be able to deal with them. On a large scale hosting system like that of Wikipedia, all this contributes only slightly to the traffic so it is not unreasonable to not take this into account in a synthetic workload.

Finally, all three benchmarks are not designed to scale. One needs to start multiple machines by hand and create tools to synchronize the start of

these individual machines. These individual benchmark processes will not coordinate with each other, making it more difficult to detect errors. E.g. if one fails, other machines will keep generating load without noticing this failure.

2.2 Web Polygraph

Web Polygraph [10] is a tool for benchmarking HTTP intermediaries like proxy servers and other web caching products. Even though this benchmark tests HTTP intermediaries instead of HTTP servers, Web Polygraph is still interesting enough to mention since the authors have taken so much care to create realistic workloads. The Web Polygraph benchmark is based entirely on synthetic traffic workloads, which are created in conjunction with industry and various research groups. The creators of Web Polygraph have tried to create realistic workloads by analyzing and using many characteristics of real web traffic, like file type and size distribution and request inter arrival times. From their research, it becomes clear that creation of synthetic, realistic workload is a research field in itself.

The Web Polygraph global architecture consists of virtual clients and servers. Clients request simulated objects from these servers. The HTTP intermediaries are placed in between. Servers and clients are glued together by using configuration files that are shared between the two.

The choice for synthetic workloads is made because for this tool's purpose, real access traces would need to be modified heavily. For example, the benchmark allows changes to parameters such as the request rate, cache hit ratios and file size and popularity distribution. The creators argue that using different (real world) traces to get different types of workloads will change many parameters at once instead of just one of the parameters. This makes analyzing performance comparisons more difficult, since different workloads are not as closely related to each other as is possible with synthetically created workloads. It is also argued that many tests do not correspond to any real trace, so using real traces would make no sense. Furthermore, using real traces for this benchmark would mean that the benchmark needs to have detailed knowledge of millions of objects. Instead, the authors have chosen to embed information about the objects requested from the server in the URLs of those objects. For example, an object type id embedded in the url identifies the properties like file size and file type of the requested object.

To vary the workload, the authors created a model that specifies a mean inter-arrival time for a Poisson stream of requests. The robots, which are similar to the previously mentioned emulated browsers, multiply this mean by the current load factor to get the correct inter-arrival time at any moment.

Unfortunately, this benchmark tool is of little use when it comes to testing systems that host web applications. Because it is aimed at benchmarking caching servers, the server side needs to host a very specific web application and web server that generates data files of different size and type based on the file name. These data files contain random data. The benchmarking application requests files of certain sizes in a certain distribution by adjusting the requested file names. Unfortunately, the web application is not representative of a typical web application. E.g. it does not serve web pages that link to each other. In addition, it does not depend on a database, making it even less realistic since most advanced web applications are very data intensive.

3 System Model

WikiBench consists of a collection of tools. The main tools are:

- **TraceBench:** A tool to process trace files for use with WikiBench
- **WikiBench:** the benchmark application itself, which can be divided into a controller and worker nodes
- **Post-processing tools:** A set of scripts to process log files produced by WikiBench and create graphics from the results

WikiBench is in the first place created as a research tool. Our goal is to create a realistic benchmark with adaptable traffic properties. We therefore benchmark a real world software application that is used extensively. This software application is MediaWiki, the software that is used by Wikipedia. Wikipedia is a collaborative, multi-language online encyclopedia. Wikipedia is based on wiki technology for storing and giving structure to information. The MediaWiki application represents the realities we are faced with today when designing software for a large scale web site with many users. This in turn means it will put realistic, high demands on a hosting system that is used to host this web application. An added advantage to using Wikipedia is that the millions of Wikipedia pages are freely available in the form of snapshots [4]. So using MediaWiki in conjunction with Wikipedia snapshots allows us to create a real world application with real world data to be benchmarked.

As was noted by other authors, like those of Web Polygraph [10], creating useful workloads with a benchmarking tool is a research field in itself. We are very grateful to have real access traces from Wikipedia. These traces, obtained directly from the WikiMedia foundation, are completely anonymized. For each request, we only have a unix timestamp, the complete url that was accessed and a 'save' flag that is set to true only if the request resulted in a page change or page creation. No more personal information than available on the publicly accessible Wikipedia website can be deducted from these traces. E.g. we do not have access to IP addresses, cookies, user account information or the contents of a POST request. On top of that, each request only has a 10% chance of being included in the traces we get. Previous research [15] has shown that these traces contain a representative sample of the workload Wikipedia gets. We therefore used these traces to create workloads for WikiBench instead of creating purely synthetic workloads like other benchmarking tools have done. This adds realism to the benchmark, since the traces we use contain all the characteristics that the real Wikipedia site has to deal with.

In this thesis we define the system under test (SUT) as the hosting system that hosts the Wikipedia data. We approach this SUT as a black box, with a URL that serves as our entry point. It is out of the scope of this thesis to define detailed requirements on a hosting system. In principle, the hosting system can apply any technique that one can think of to improve overall system performance. The only "common-sense" requirements we put on the system is that is is accessible through HTTP, exactly like the MediaWiki software, and that it gives the same responses in identical situations. If the SUT would be run side by side with a vanilla MediaWiki installation, we expect the exact same replies when doing identical GET and POST requests on both servers. There is one exception to this rule: we removed a check for duplicate edits and CSRF attacks ([5], [17]) from MediaWiki. We worked around this check by commenting out a single line of MediaWiki code. Without this check, we can post anything we want to a wiki page with a single http POST request. If we would leave this security check in the code, we would need to first request the edit page, parse that page to obtain a number of hidden form elements and then perform the POST request. The hidden field, a secret token that is user specific, is used to prevent CSRF attacks in which a logged on user might POST data to Wikipedia without knowing it while he is visiting a malicious web site. Because a malicious site can not guess or calculate this secret token, it can never perform a valid POST request without the user noticing it. Leaving the check in place would also increase the chance of an edit conflict. While requesting and parsing the edit page, a time window is introduced in which another thread might request that same page. Because MediaWiki uses timestamps to check for edit conflicts, we will now have to solve an edit conflict between two threads, introducing even more complexity.

3.1 Requirements

Wikipedia has a number of characteristics that make it ideal to be used in a distributed web application benchmark. First of all, there is lots of data. Just the English wiki, without user and talk pages, contains more than 7 million articles in the snapshot taken on October 7, 2008. This can be hosted on one single server, but that server will only be able to sustain fractions of Wikipedia's actual traffic. Wikipedia receives high numbers of requests per second. As of this writing, peaks of 50,000 to 60,000 requests per second are no exception. So, even with our 10% sample of the access traces, we can generate high loads up to about 5000 requests per seconds. The requirements we put on WikiBench obviously need to match these numbers. This subsection describes the main goals and requirements we have put on

WikiBench.

3.1.1 Representative of real world traffic

What constitutes traffic realism? A good synthetic traffic generator can provide very realistic traffic. We certainly recognize and applaud the efforts that the writers of Web Polygraph have put into creating realistic traffic generators for their benchmarking tool, for example. WikiBench is meant to provide a means of testing hosting systems that host a large scale wiki site like Wikipedia. For this reason, we define realistic traffic to be traffic that matches, as closely as possible, the traffic that a large scale wiki engine receives. Some important aspects of such traffic can be found in [15], in which Wikipedia traces are statistically analyzed. We identify the following traffic properties that we consider important and which we want to retain:

- The interarrival times between requests: Wikipedia traces show visitors as possion arrivals, e.g. arriving randomly spaced in time
- The read/write ratio of wiki pages: this ratio tends to be high, e.g. there are many reads and not that many writes. This differs greatly between countries though as can be seen in [15]
- The ratio of static/non-static file requests: it is shown in [15] that only a small percentage of requests is for pages, most requests are for images, binaries and other static files
- The distribution of page popularity: There are four very popular pages, like the Main Page and style sheet page, then a large part of the pages roughly follows a Zipf distribution and finally a large part is accessed very infrequently
- The distribution of page save and update operations, which follows a Zipf distribution. We can also see that more popular pages tend to be updated more frequently
- Strong load variations on the page level, initiated by events taking place in the world. These load variation are very typical for Wikipedia and are a challenge to decentralized hosting systems
- A considerable amount of requests for non-existing pages and files, which obviously add realism to the traffic.

3.1.2 Usable for both large and small scale benchmarks

WikiBench needs to be useable for both small and large scale benchmarks and anything in between. This requirement has direct consequences for the workloads that are used by WikiBench. The large amounts of traffic we see on Wikipedia somehow need to be scaled down while maintaining important traffic properties as stated above.

3.1.3 Reproducible results

We want results to be reproducible. When we process trace files to reduce traffic intensity, we want to consistently drop the same pages when running the processing tools multiple times. This means that when we create a workload file twice with the same settings, it should have the same contents.

3.1.4 Highly scalable

WikiBench should be able to generate traffic intensities matching those of Wikipedia, so it needs to be scaleable.

3.1.5 Adaptable traffic characteristics

Besides offering a realistic workload, we also want to be able to change traffic characteristics, e.g. we want to be able to create global flash crowds, change the read/write ratio of wiki pages or adapt other traffic properties by altering the workload file. Although such changes will lower the realism of the traffic, they can be very useful when testing hosting systems.

3.1.6 Portability

We want WikiBench to be portable, e.g. it should not be tied to one operating system. We also want WikiBench to literally be portable — one should be able to download WikiBench and associated files and run it. WikiBench therefore includes workload files and a database snapshot, since we can not be sure about the lifetime of snapshots hosted by WikiMedia. By including this data WikiBench will be usable "out of the box".

3.2 WikiBench design

Most of the WikiBench code is written in Java. This choice is made because of portability and ease of development. The use of Java for high performance computing requires conservative use of certain libraries. We refrained from

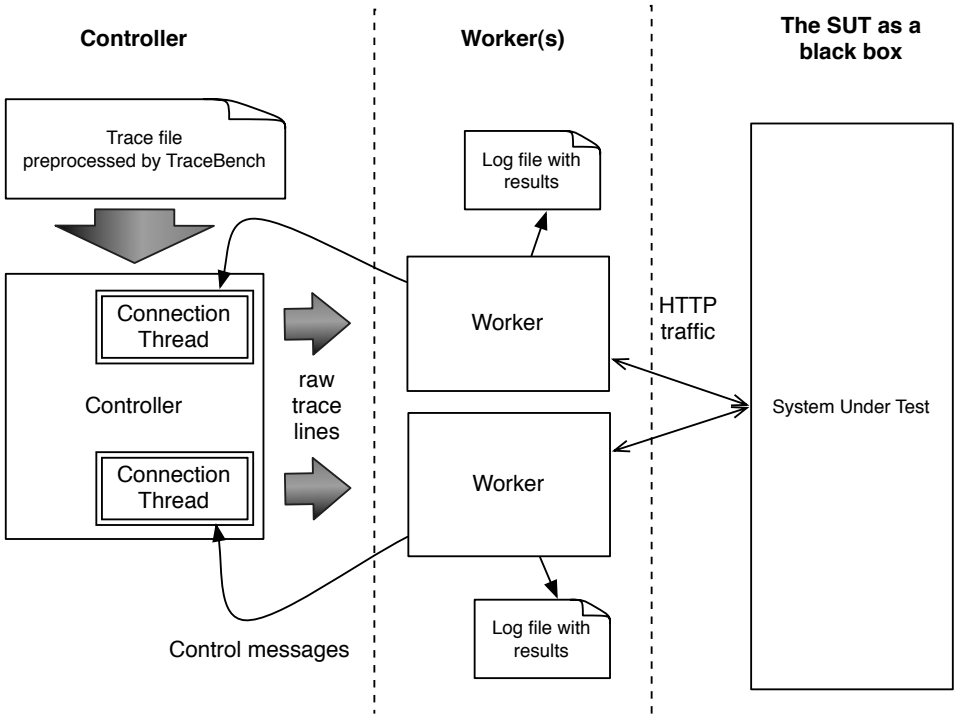

Figure 1: WikiBench design and data flow

using Java's built-in URL and HTTP libraries. Instead, we use HttpCore 4.0 [9] from Apache's HttpComponents project. HttpCore is the most low level library from this project. It is lightweight and it allows us to customize requests to a level that is not available in Java's standard http libraries. HttpCore is in beta as of this writing, but the library calls have been frozen and the library turned out to be very stable during our experiments — its small memory footprint combined with a very moderate system load work out very well for WikiBench. We also refrained from using Java RMI for connectivity between multiple WikiBench machines. Java's object serialization is an important factor in the speed of RMI. For small or sporadic communications this serialization is not a problem. For sending many gigabytes of trace lines however, it is. Since we only have to send over strings of text, we use TCP sockets instead.

As shown in Figure 1, WikiBench consists of one controller and one or more worker nodes. The WikiBench core can be started either in controller mode or in worker mode. The tasks of the controller are to distribute trace lines to the workers and coordinate work. By using simple plain text control messages the controller communicates with all connected workers. The

worker nodes iteratively get a trace line from the controller, parse the URL and timestamp from that line, and perform a GET or POST request to the system under test.

At the controller, there is an object called the TraceReader which has exclusive access to the trace file on hard disk. Inside the controller, one thread called a WorkerConnection is created for each worker that connects. The WorkerConnection thread constantly requests trace lines from the TraceReader object and blindly writes these trace lines into the socket at its maximum speed. Once a socket blocks because buffers are filled, the thread blocks until the worker at the other side of the socket has read enough lines. With this setup, each worker runs at the maximum speed it can sustain. After writing a line to the worker, the WorkerConnection checks if a message from the worker is available for reading. This way a worker can notify the controller of important events. The only event currently implemented is the situation in which a worker has failed to meet its deadlines. More on this follows shortly hereafter.

Before the benchmark starts, the controller sends out a start time to the workers. This start time is based on the real time clock, so it is important that all participating nodes have their time synchronized. This can be done with high accuracy by using the Network Time Protocol, available on practically all operating systems. This is important, since all the workers need to start at the same time. The timing of a HTTP request is calculated based on this start time and the first request time that is found in the trace file as follows. Each worker receives the time of the first request in the trace file. It subtracts this time from all the timestamps it receives, so all timestamps are made relative to this initial timestamp. The result is added to the starting time. In a formula:

$$AbsoluteRequestTime = AbsoluteStartTime + (timestamp_c - timestamp_i)$$

In this formula, $AbsoluteRequestTime$ is the absolute time at which the request has to be performed. $AbsoluteStartTime$ is the time at which the benchmark was started, this is the time the controller send to all the workers. $timestamp_c$ is the timestamp found in the current trace line and $timestamp_i$ is the initial timestamp found in the very first line of the trace file.

On the worker side, there is a fixed number of threads called FetchThreads. These FetchThreads read and parse a trace line and use Java's sleep() method to wait for the appropriate amount of milliseconds to pass before performing the request. It can happen that the workers are not able to keep up with the pace. There are at least two situations that might lead to such a problem. When the supply of trace lines is too large, the worker will at some point

reach the maximum number of threads created at the start. This amount of threads is fixed and started before the actual start of the benchmark, because dynamic thread creation may negatively impact performance and correctness of measurements. Another situation in which missed deadlines may occur is when the SUT goes down. The maximum time out value can cause a situation in which all threads at some point will be waiting for the SUT to respond. E.g. when the timeout is set to one minute and there are 500 threads, the system will be able to generate a maximum request rate of about 500 requests per minute because each thread can wait for a maximum of one minute to get a reply from the SUT. A missed deadline can be determined easily because the calculated $AbsoluteRequestTime$ will be in the past. In this case, the worker sends back a message to the controller and the controller will abort the benchmark since its results will no longer be reliable — the timing of requests no longer matches that of the workload file.

From the above it should be clear that each request is bound to strict timeouts. The default timeout can be changed by giving command-line parameters to the workers. If the SUT can not keep up with the amount of requests, WikiBench will keep generating requests. This is a fundamental difference with other benchmarking tools like TPC-W. This way of testing matches reality: people generally do not wait for a system to become responsive again.

The request type, POST or GET, is determined by the trace line. If the trace line ends with a dash, there is no post data, meaning it is a GET request. Performing a GET request is easy. WikiBench fetches the page, calculates the page size and drops the fetched data directly after the request. If there is post data in the trace line, this data is deserialized and posted to the URL that is in the trace line. This POST data is added by TraceBench, which is discussed further in Section 3.3. Doing a POST request is not trivial. First of all, the HttpCore package we use has no helpers for POST requests, so we need to construct a proper HTTP POST request ourselves. As we already mentioned, we removed the check for duplicate edits from MediaWiki, which allows us to directly POST to a wiki page. Without this change, we would need to request the edit page first and then parse a number of properties from the form on that page, like timestamps. MediaWiki uses this information to detect edit conflicts and to protect itself against automated page vadalism. Since we only have a 10% sample of the Wikipedia traffic it is very likely that this edit page has not been requested yet, which would force us to insert requests for edit pages that are not in the workload file. Removing these checks prevents us from taking such measures: we can now do POST requests directly. An added advantage is that we do not have to solve edit conflicts, which may be introduced when multiple threads try to edit the same page.

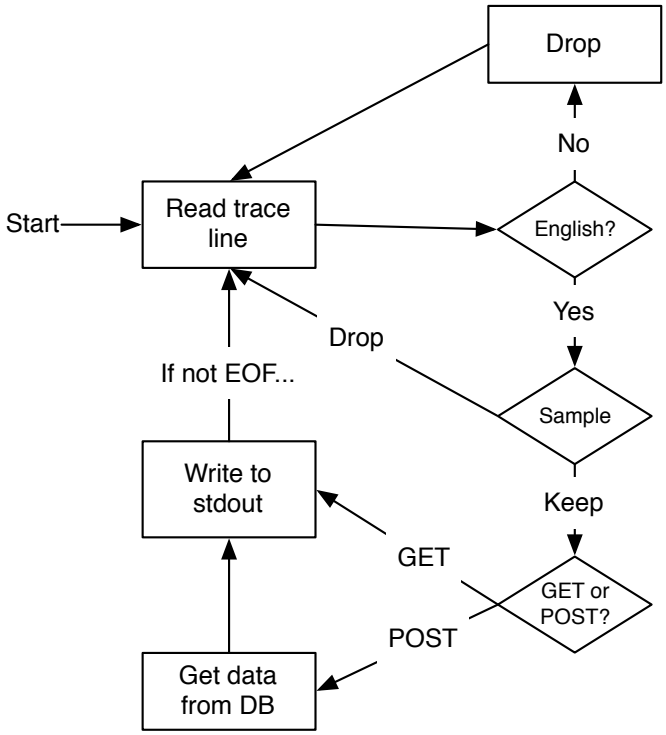

Figure 2: TraceBench loop

The time period between requesting the edit page and performing the action edit can be quite large — humans tend to be slow — therefore MediaWiki uses the timestamp, hidden in the edit form, to check if another user has changed the page in the meantime. Section 6 offers a better solution for cases where full traces are available.

3.3 TraceBench Design

TraceBench is our trace processor, a Java application that reads trace lines from standard input and writes processed trace lines to standard output. TraceBench is needed to reduce traffic, which it can do by using sampling. TraceBench can also add POST data, obtained from a database, to the trace file. TraceBench takes three command-line parameters:

- the reduction permil, a number from 0 (drop none) up to 1000 (drop all)
- a JDBC link to the Wikipedia database

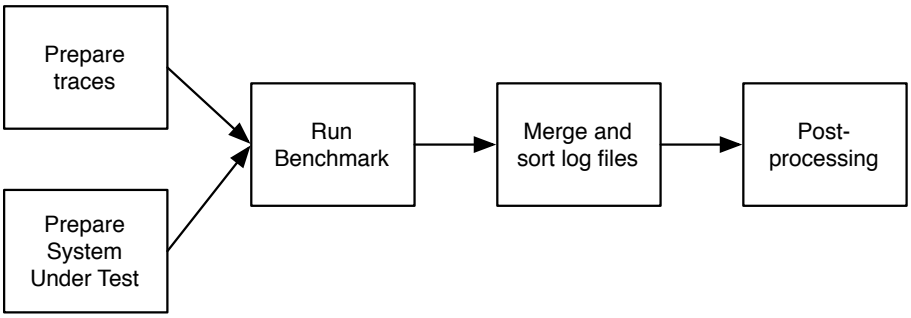

Figure 3: WikiBench workflow

- the sampling method

The reduction number expresses the amount of trace file to be removed in permil (‰). The JDBC link is needed to supplement the resulting trace file with POST data. This POST data currently consists of the latest version of a wiki page and the time of the latest edit. Future versions might get specific revisions from the database. That way, WikiBench would be able to "replay" the trace files exactly as it happened in the past. TraceBench fetches this data directly from the provided database, uses URL encoding to serialize the data so it can be added at the end of the request in the trace file. Finally the sampling method determines how traffic intensity is reduced. This is explained in Section 4.

The architecture of TraceBench is simple and consists of a basic loop as can be seen in Figure 2. The loop starts by reading a line from standard input and checking if the trace line corresponds to a request to the English Wikipedia (`en.wikpedia.org`). If not, the line is dropped and we start over. If it is, the URL in the trace line is parsed further. The next step, based on this URL, is to sample the trace line. There are two sampling methods. The user can specify the method to be used on the command line. These methods are discussed in detail in Section 4. If the sampling method decides this trace line is a keeper, we determine if it is a POST or a GET request. If the request is found to be a POST request, it means that this is an edit or creation of a wiki page. In such cases TraceBench fetches the content and timestamp from the database, serializes the data by using URL encoding and adds it to the end of the trace line. We finish by writing the newly created trace line to standard output. This loop repeats until end of file is reached.

3.4 WikiBench Workflow

Figure 3 show the typical workflow of a WikiBench benchmark. We will now look at each step in detail and give a general idea of all the steps that need to be taken in order to run a benchmark.

3.4.1 Trace File Preparation

Trace files, as obtained from the Wikimedia Foundation, are not completely sorted by timestamp, contain unnecessary data like request ids — each request is numbered besides having a timestamp — and have several lines with missing or scrambled data. Typically, one starts by sorting the trace file by timestamp and removing the ids that were added in front of each line. This can be done with regular tools, like gawk and the unix sort tool. After this, the trace file can be fed to TraceBench, which is robust enough to detect further problems like missing data or empty lines. The vast amounts of data make the preparation of the trace files more difficult than it might seem. There are limits to how large a file you can sort with the unix `sort` command, for example. Processing a trace file containing 24 hours worth of data can be done on a single PC, but beyond that you will need to use specialized tools. Since the trace files are sorted to a great extent already, one might think of a sort tool that only sorts within a time window instead of looking at the entire data file. We have not created such a tool, since we could do without so far. Since these trace files are not generally available, we include ready to use trace files with WikiBench.

3.4.2 Configuring The Initial System State

WikiBench can be used with any system that implements the same interface as Wikipedia. For our initial benchmarks, we have setup a Wikipedia mirror using the same software components as Wikipedia: an Apache 2.2 web server, PHP5, MySQL 5 and the MediaWiki software. We have installed Xcache[18], a common PHP extension that caches php opcode, to give our system a slight speed bump. Setting up a Wikipedia mirror on a server is not straight forward. The amount of data is large, so importing it takes much time. For this research, we have used the Wikipedia snapshots from October 7, 2008. There are roughly 7 million wiki pages in this snapshot, not including user and talk pages. The XML dump containing these pages can not be parsed with regular tools, since these tools tend to create a complete tree of the XML data. There are a number of tools that were made specially for the purpose of converting Wikipedia dumps to SQL statements. More info on these tools can be found on the MediaWiki site [1]. Due to limited resources, we have

not included Talk and user pages. Even without content, this data exceeded the available disk space on our system. TraceBench does not include requests for Talk and User pages, but can be adapted to do so with little effort.

3.4.3 Running the Benchmark

Once the System Under Test is up and running, it is necessary to first do a few partial benchmarks to "warm up" the system. It helps if MediaWiki, Apache, MySQL and your filesystem are prepared. Apache can for example increase the amount of processes, MediaWiki has an object cache that gets filled with cached objects and MySQL can improve performance by loading index files into memory.

A partial benchmark is enough, since the page level sampling reduces the amount of pages considerably. So after about ten minutes, the SUT has served a large part of all the pages that are in the trace file. During warm up, it is not uncommon for the SUT to become overloaded.

3.4.4 Post-processing

Since each worker logs results to its own file, these files need to be merged and sorted after the benchmark. This is, for now, a manual process. With an `awk`-script we are able to extract data from the final log file. This script takes a time period in seconds as a parameter and calculates the median response time, the number of requests and the number of time outs that occurred in each time period. By using a few simple `gnuplot` configuration files, these results can be plotted into a number of images.

4 Workload Creation

By now, it should be clear that WikiBench uses trace files to create benchmark workloads. We believe the usage of files has a number of advantages:

- the workloads can be adapted without changing WikiBench code, reducing the chance of code errors
- preproduced workloads reduce the system load while running the benchmark
- the ability to fetch POST data from the wiki database before running the benchmark, in order to supplement the trace files
- the possibility to easily alter the workload, e.g. by filtering out requests using a one line `grep` or `awk` command

We think these advantages weight out the disadvantages such as the required storage space for trace files and the introduction of an extra step in the benchmark process.

4.1 Changing the Request Rate

The use of access traces however requires us to be able to reduce or increase traffic intensities. Increasing traffic intensity can be done by combining multiple periods of trace file into one. We do not support this yet — the trace files that are available currently have more traffic than our System Under Test can handle. Reducing traffic is more difficult. We will discuss three methods and propose our method, which consists of a hybrid approach of two of these methods.

4.1.1 Time stretching

With time stretching, we mean multiplying the inter-arrival times between requests with a constant factor. This way, the overall request rate goes down while all but one properties of the traffic are preserved: only the inter-arrival times change. A problem that quickly arises with this method is that the minimum time unit in our trace files is a millisecond. There can be 10 or more requests in a millisecond, which leaves us without a time interval between most requests in the trace file. In short this would be a good alternative if we would have complete traces with timestamps of microsecond precision.

4.1.2 Request Level Sampling

Since our traces are already a 10% sample of the complete traces, sampling this even further seems to be a natural choice. This alternative preserves the inter-arrival times as they are found in the trace. We end up with an even smaller percentage of the actual traces. Such a trace would however still be a realistic traffic simulation, since the trace could represent the traffic that part of a larger hosting system would receive when a load balancer is put in front of the complete system. The formulas used for this type of sampling look as follows:

- Pick a random number N_{rand}, with $0 < N_{rand} < 1000$
- If $N_{rand} < R$ drop request, where R is the reduction permil
- Otherwise, the request is kept

The downside of sampling the sample is that we damage the distribution of page popularity, in terms of amounts of requests. Wikipedia has millions of pages and there exists a very specific distribution of page popularity, as shown in [15]. A small part of all the pages receive a majority of the total traffic. There is a long tail of pages that receive considerably less visitors. A problem that occurs when removing requests at random is that the pages in this tail get either overrepresented or underrepresented. Intuitively explained: a page that is visited only once in a month can have a large impact on a system because it might not exist in any kind of cache. The cost of building and showing such a page is large. If such a request gets dropped, the impact is much larger than dropping a request for the Wikipedia home page, for example, which is one of the most requested pages and as a result one of the cheapest ones to serve to a visitor since it will reside in many cache servers. Another way to explain this is that dropping 1 request of 1000 requests to a page has less impact than removing the only request to a low-traffic page, which effectively removes the page from the system altogether. The second downside of sampling is that we may lose page creations and updates at random. Updates are a challenge for a hosting system since caches will be invalidated. We would rather not lose them. Loosing one page creation has a lot of impact too, because subsequent reads will fail if we don't take special measures. We could therefore think of a sub-alternative: sampling, keeping all the updates and page creations. The obvious advantage of this alternative is that we do not lose our page updates and creations. The downside however is that we change the read/save ratio, so we lose some of the realism. This last method is one of the two modes that can be used in TraceBench.

4.1.3 Page Level Sampling

Instead of sampling requests, we can also sample pages. By this we mean that a subset of the total number of pages is kept and other pages are dropped. This corresponds to a realistic scenario in which pages are partitioned over a system in which each part is responsible for maintaining and serving a subset of the pages.

We can implement this type of sampling by calculating a hash of the page name. The following formulas determine if a page needs to be kept or dropped:

- Calculate a cheap hash from the page name: H_p
- If $H_p \bmod 1000 < R$ drop request, R is the reduction permil
- Otherwise, the request is kept

By sampling pages instead of requests, we do not have the problems of over- or underrepresentation of infrequently accessed pages. We consistently keep or drop certain pages, based on their name. This method reduces the amount of pages known to WikiBench. In mathematical terms this alternative of sampling creates a subset of the full set of pages. There are three problems that arise with this alternative. First, there are many templates and pages that are included in other pages, called transclusions by Wikipedia. Removing a template or page that is included elsewhere will break those pages. So when using this alternative, either the complete database needs to be kept, or such pages need to be identified and retained specifically. Another problem with this approach is that there is a chance that a highly popular page, like the Main Page, will be dropped. Such pages need to be excluded from this sampling method explicitly. The last problem is that when removing just pages, the amount of requests for static files like style sheets and template images will be disproportional to the amount of requests for pages. For this reason, we propose a hybrid approach.

4.2 A Hybrid Approach

We want the best of both worlds, so we combine the regular sampling of requests and our page sampling alternative. Let us first divide the traces into two types of request: page requests and non-page requests. A page request is defined as a read or a write operation on a page. All other requests are the non-page requests. Sampling at the page level gives us a fair distribution of popular and less popular pages, thanks to the use of a hash function. It will

however create an imbalance in the page/non-page ratio. In other words, there will be way to many requests for static files and such. Since most requests for static files are the result of a page request however, these request types are related. E.g. a user requests a page and as a result of this several images and style sheets are also requested. We can use regular sampling for all these non-page requests, removing at random a equally large percentage of requests in order to restore the balance between page and non-page requests. The exact percentage to remove is easily derived from the reduction ratio the user specifies. Because the static file requests are related to page requests, the distribution and density follows the page requests closely. There are few static files, so we do not risk over- or underrepresentation of static files.

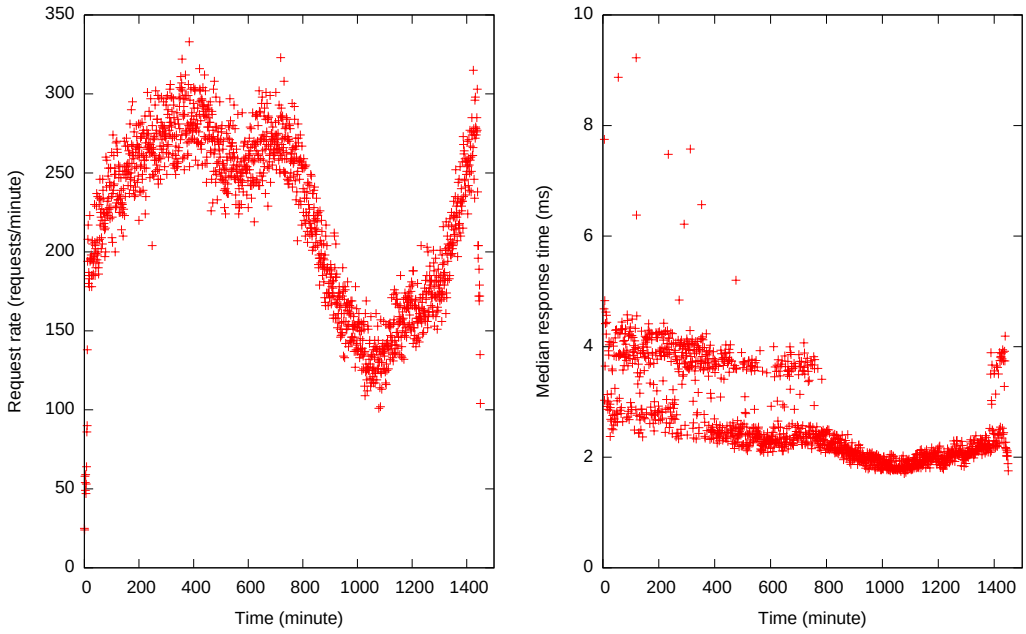

Figure 4: A 24h benchmark using 0.05% of Wikipedia's traffic

5 Benchmark Results

Our first benchmark is done on a stand alone server running Apache, MySQL and PHP. It runs a vanilla MediaWiki installation, that is configured such that it mimics the URL structure of Wikipedia. The only optimization we have done is installing the XCache PHP opcode cacher. The system under test is a PC with 2GB of memory and an AMD Athlon 3200+ CPU. WikiBench and the SUT are connected through a dedicated 100mbps router.

We have created a number of trace files using TraceBench with different traffic intensities, ranging from 0.5% to 2.0% of the original 10% sample of the Wikipedia traces. The system under test was able to sustain only the 0.5% trace. Since taken from a 10% sample, this trace contains only 0.05% of the actual Wikipedia traffic. We used the hybrid sampling approach for this benchmark.

While doing this benchmark, the importance of warming up the system has become very clear. The first run of a 0.05% trace overloaded the server, causing numerous timeouts and HTTP responses with "500 Internal server error" status codes. The internal server errors were mostly caused by an unresponsive database. Consecutive runs cause moderate to heavy loads.

Figure 4 shows a typical day of traffic on the English Wikipedia site. On the left the number of requests per minute is plotted against time. On the

right, the median response time per minute is plotted against time. These figures show a clear relation between the request rate and the response times, since the high traffic periods show a higher median response time.

While running our benchmarks, we noticed a clear distinction between the response times of wiki pages and non-wiki pages. Figures 5 and 6 show the median response times in separate graphs. From Figure 5 it becomes clear that median response times of the wiki pages are significantly higher than those of non-wiki pages and files. It is difficult to see a difference between response times in high and low traffic periods. At around the 550th minute a significant drop in the median response times can be seen. This drop is caused by the system being completely overloaded. The low response times consist of pages that returned an "Invalid server error": the server notices it can't handle the traffic and, without the need for disk seeks and database queries, it serves up a very simple error page.

Figure 6 shows a consistently low response time for non-wiki pages, like image files and other static documents. These easy to cache documents, once stored in cache memory, can be served very quickly. The 2 ms response times are mainly caused by network transfer and processing time inside the web server.

As discussed before, there exists a clear relation between a wiki page and non-wiki documents. Each request for a wiki page is followed by several supporting documents like images and style sheets. Because of this relation, non-page requests outnumber the page requests. This is the reason why, in figure 4, you don't find the high response times from figure 6. Because we are using medians, the median response time generally is very close towards the low response times of non-page requests. This is also the reason why we have split up Figure 4. There is a lot of extra information that would otherwise not be visible. In this regard it should be noted that a visitor's perception of loading speed is greatly dependent on the speed at which the HTML pages load. Therefore Figure 4 gives a more realistic view of perceived speed compared to Figure 5.

These figures show why Wikipedia can be so successful with relatively simple ways of scaling their hosting setup. Since most pages are updated only occasionally, the caching of wiki pages introduces a huge performance gain. The non-wiki page requests are even easier to cache; cached objects of this type can have long life times. The hosting setup of the WikiMedia Foundation consists of a large amount of web caching servers (Squid). The use of web caching servers in this case greatly reduces the load on the web and database servers. However, the usage of traditional database servers will at some point limit Wikipedia's growth. The sheer number of articles that continues to grow is increasingly difficult to store in a traditional datbase and

at some time a more scalable solution will be required, such as a completely decentralized setup [15] or the use of distributed key-value stores like Dynamo [6] to store page revisions.

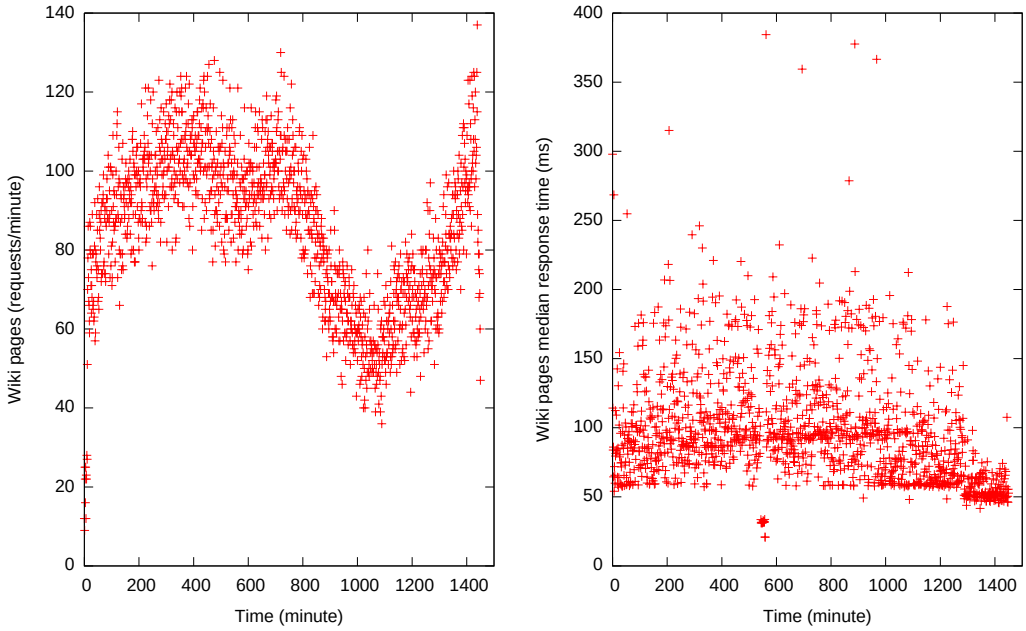

Figure 5: The 24h benchmark, only considering wiki pages

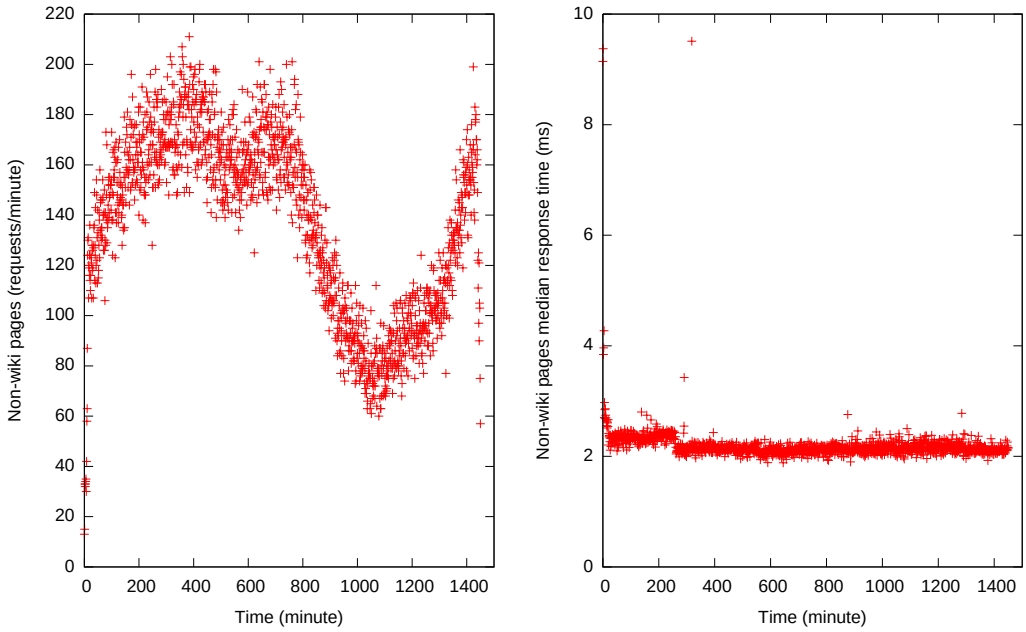

Figure 6: The 24h benchmark, only considering non-wiki pages

6 Future Work

WikiBench, in its current form, is a very basic tool. There are several extensions and improvements with can think of, but did not have time to implement. We will discuss a few of such extensions.

6.1 Scaling up traffic and Flash Crowds

So far we have focussed on how to reduce traffic intensities. Increasing traffic intensities can be done too, for example by combining multiple days worth of data. Another possibility is the addition of synthetic traffic. It would reduce the traffic realism, but it could be very useful for testing purposes.

It has been shown in [15] that Wikipedia does not receive global flash crowds. Especially by combining the traffic of multiple different time periods we could create very realistic flash crowds with relative ease.

6.2 Adjust read/write ratio

Adjusting the ratio between page reads and writes can be especially interesting when testing the impact of updates on a hosting system. More updates will cause more load on the wiki database. More importantly, more updates will cause cache servers to be less effective, since pages will age quicker, thus increasing the load on the web servers and databases. A setup that makes use of many caching servers, as is done to host the real Wikipedia site, would be considerably less effective if it would be hosting a wiki site with many updates.

Changing this ratio would require the injection or removal of update requests in the trace file. In this case, one needs to take into account the hybrid sampling approach, in which only a subset of pages can be injected. Furthermore it would be more realistic to also balance the static/non-static ratio of document requests, for example by adding requests for static files in the area where updates are inserted.

6.3 Adjust the distribution of page popularity

In a distributed system where wiki pages are spread over multiple decentralized nodes, as it is proposed in [14] for example, a change in the distribution of page popularity would require the hosting system to recalculate resource assignments per node.

Since a wiki can be used for all kinds of purposes, it can be very interesting to change this distribution to match the expected usage of a system. It

can also be interesting to change this distribution while in the middle of a benchmark, since the hosting system will have to dynamically recalculate the assignment of resources for many pages.

6.4 Indication of realism

A desirable addition to TraceBench would be the output of a numerical indicator of traffic realism. Such an indicator can give the user a quick way of determining traffic realism. This is especially useful when traffic is altered in more than one way, like reducing traffic and adjusting the read/write ratio.

6.5 More advanced edits

Right now, we have altered the MediaWiki code in such a way that it does not check for edit conflicts. Although this constitutes a very small change it is not the most desirable solution and there are in fact better ways. Instead of randomly sending trace lines to the worker that asks for them, we can also consistently assign page names to hosts. This fixed assignment can be done easily with a hash and a modulo operation, since the amount of workers is fixed. Each worker can now keep state and remember the last edittime of each page. This solution works only if we have full traces, since we will need to fetch the edit form before submitting a page edit or creation with the proper timestamps included.

7 Conclusion

The importance of web applications has increased over the past years, while only a limited amount of benchmark tools emerged to test the systems hosting these web application. These tools have important limitations, which is why we introduced WikiBench, a scalable web application benchmark tool based on Wikipedia. WikiBench has advantages over other benchmark tools in the areas of scalability and realism of the generated workloads. WikiBench uses Wikipedia's MediaWiki package, Wikipedia database snapshots and access traces obtained from the Wikimedia Foundation.

We defined a number of important requirements for WikiBench, like realism and scalability, and worked towards a solution that satisfies these requirements. We showed the design and implementation of WikiBench, in which a single controller node distributes work to several worker nodes and coordinates the benchmark. TraceBench, our trace file processor, is able to reduce traffic intensities while maintaining important properties like inter-arrival times and the read/save ratio of wiki pages. Key to maintaining these properties is a combination between regular sampling and page level sampling, which uses page name hashes to include only a subset of the entire set of wiki pages in the generated workload files.

Our initial benchmark show a 24 hour period of Wikipedia traffic. The effects of increasing and decreasing traffic intensities is shown by plotting median response times, showing a clear relation between traffic intensity and response times from the system under test.

We proposed a number of improvements to WikiBench and TraceBench, such as a more advanced way to edit and add wiki pages. We also proposed ways to alter the workload in order to change traffic properties, such as scaling up traffic by merging multiple time periods of data and adjusting the read/write ratio by removing or injecting page edits. By changing such properties important aspects of a hosting system can be tested. We leave these improvements for future work.

References

[1] Mediawiki, a free software wiki package. `http://www.mediawiki.org`.

[2] Rubbos: Bulletin board benchmark. `http://jmod.objectweb.org/rubbos.html`.

[3] Rubis: Rice university bidding system. `http://rubis.objectweb.org/`.

[4] Wikimedia downloads. `http://download.wikimedia.org/`.

[5] R. Cannings, H. Dwivedi, and Z. Lackey. *Hacking Exposed Web 2.0: Web 2.0 Security Secrets and Solutions*. McGraw-Hill Osborne Media, 2007.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[7] D. F. Galletta, R. Henry, S. Mccoy, and P. Polak. Web site delays: How tolerant are users. *Journal of the Association for Information Systems*, 5:1–28, 2004.

[8] T. Groothuyse, S. Sivasubramanian, and G. Pierre. GlobeTP: Template-based database replication for scalable web applications. In *Proceedings of the 16th International World Wide Web Conference*, Banff, Canada, May 2007. `http://www.globule.org/publi/GTBDRSWA_www2007.html`.

[9] HttpCore. A set of low level http transport components that can be used to build custom client and server side http services. `http://hc.apache.org/httpcomponents-core/index.html`.

[10] A. Rousskov and D. Wessels. High-performance benchmarking with web polygraph. *Softw. Pract. Exper.*, 34(2):187–211, 2004.

[11] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. GlobeDB: Autonomic data replication for web applications. In *Proc. of the 14th International World-Wide Web Conference*, pages 33–42, Chiba, Japan, may 2005. `http://www.globule.org/publi/GADRWA_www2005.html`.

[12] S. Sivasubramanian, G. Pierre, M. van Steen, and G. Alonso. GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, June 2006. http://www.globule.org/publi/GCBRCDWA_ircs022.html.

[13] W. Smith. Tpc-w: Benchmarking an ecommerce solution. *Whitepaper, Transaction Processing Performance Council*.

[14] G. Urdaneta, G. Pierre, and M. van Steen. A decentralized wiki engine for collaborative wikipedia hosting. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, Mar. 2007. http://www.globule.org/publi/DWECWH_webist2007.html.

[15] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[16] Z. Wei, J. Dejun, G. Pierre, C.-H. Chi, and M. van Steen. Service-oriented data denormalization for scalable web applications. In *Proceedings of the 17th International World Wide Web Conference*, Beijing, China, Apr. 2008. http://www.globule.org/publi/SODDSWA_www2008.html.

[17] WikiMedia. Mediawiki 1.3.11 release notes, February 2005. http://sourceforge.net/project/shownotes.php?release_id=307067.

[18] XCache. A fast, stable php opcode cacher. http://xcache.lighttpd.net/.