ON THE RESPONSE TIME OF A SOLR SEARCH ENGINE IN A VIRTUALIZED ENVIRONMENT

BRIAN BOUTERSE
Department of Computer Science, North Carolina State University, Raleigh, NC, USA
email: bmbouter@ncsu.edu

HARRY PERROS
Department of Computer Science, North Carolina State University, Raleigh, NC, USA
email: hp@ncsu.edu

This paper describes the results of measured response times of a full-text search service running in a virtualized environment. Using a monitoring tool, we observed the performance of the lulu.com Solr search engine in order to identify a relationship between the CPU and memory allocated to the Solr VM, the arrival rate of queries, and the 95th percentile and the mean of the query execution response time. Based on collected data, we then fitted an analytic expression which relates the 95th percentile and mean of the response time to the CPU allocated to the VM and arrival rate of queries. These expressions can be used for capacity planning of the Solr search cluster at Lulu.

1. Introduction

We describe the response time of a web-based service running in a virtualized environment. Specifically, we report on the response time of the lulu.com [1] search engine which is based on Solr, as a function of CPU and RAM allocated to the VM containing the Solr and its internal caches for various arrival rates of queries. We use the 95th percentile of the response time as our metric of service performance. Percentiles provide a better metric than the commonly used average, which does not reflect well the variability of the observed metric. For comparison purposes, we also provide the mean response time. Based on the results obtained, we developed an expression that predicts the 95th percentile and mean of the response time as a function of the CPU allocated for a given arrival rate of queries. This expression can be used for planning purposes.

To the best of our knowledge the response time of Solr running in a virtualized environment has not been reported before in the open literature. In order to measure the response time, we developed a monitoring tool with a network protocol analyzer that calculates the response time of Solr queries from the TCP packets that are generated for each query. There are many network performance analyzers, such as TPC-App [2], Wireshark [3], and Snort [4], but none of these provide this feature. (Due to lack of space we cannot give a comprehensive comparison.)

The remaining of the paper is structured as follows. In the following section, we describe the testbed and tools used to conduct the performance measurements. In section 3, we describe the test methodology and the results are presented in section 4. Finally, section 5 provides the conclusions.

2. The Lulu Search Service and Testbed

The search service on the Lulu [1] homepage is supported by a load-balanced cluster of search servers which handle all search queries and return the results to the web front-end. The web service tier submits requests using a GET style HTTP 1.1 request with the query and number of arguments to port 8080. After the database is searched, the HTTP response status is 200 OK and its body contains an XML document describing the query results. The Solr servers backing the Solr cluster in the Lulu architecture, are virtualized and currently execute as VMware ESX based virtual machines. These Virtual Machines have adjustable CPU and RAM, and they have virtual hard disks located on the local hard drive of the hypervisor.

The software stack of a Lulu search server has five components: the virtual environment, the Linux Operating System, the Java runtime environment, Jetty webserver, and Solr, an enterprise class, Java based search software package. Jetty is an open-source, Java based web-server which executes the Solr application as a java servlet. The Solr index containing the records to be searched is stored locally inside each Solr server's virtual disk, which physically resides on the local disk of the hypervisor. For different resource configurations, the CPU and RAM allocated to a Virtual Machine environment are modified. The Jetty runtime environment also has its RAM size adjusted with each test to allow the Solr environment to take advantage of the additional memory. CPU and RAM resources both have their allocated amount reserved to ensure they are not disturbed by other workloads on the hypervisor.

To benchmark the performance of the Lulu search service without interrupting the Lulu production operation, a separate testbed was created. The search service benchmarking testbed included three main components: the Lulu Solr search service virtual machine, a customer request generator, and a traffic collector. The Lulu Solr Virtual Machine (searchVM) is a snapshot of a production search server at lulu.com from 9/3/2008, and was duplicated into the testbed by copying the virtual disk (.vmdk) and the definition files (.vmx).

A customer request generator was configured to create a query load for the search VM that recreates the loads experienced at the Lulu web front-end. We assume that the customers arrive at the web front-end according to a Poisson process with an arrival rate λ. Validating this assumption in real-world environments is beyond the scope of this paper. To generate customer requests according to a Poisson arrival rate, the Tsung load testing tool was used [7]. Tsung simulates user arrivals according to a Poisson process to test the scalability and performance of IP based client/server applications. The queries for the testbed were obtained by parsing a large set of real-world queries obtained from actual Lulu production server HTTP logs from September 2008. There are 10 results for each query, and Solr is asked to sort them in descending order.

Physically the testbed consists of one 4948 Ethernet switch with three IBM HS20 Blades inside a Blade Center where each Blade has 2x3.6 Ghz processing power, 16 GB of RAM, and 2 copper 1GE NICs. The switch provides connectivity between the query generator and the Solr search server and it duplicates all query-solr traffic onto an interface that is connected to the traffic collector. This was implemented by configuring a Switched Port Analyzer (SPAN) port, which duplicated the traffic flow between the Tsung query generator and the Solr search server. This duplicated all the query transactions sent to the traffic collector machine and stored all traffic into a packet capture file for offline analysis. In order to ensure that the testbed is isolated from any other network traffic on the switch, two private VLANs, A and B. The SPAN port was configured to duplicate all traffic from VLAN A to VLAN B, and the traffic collector was configured to record all traffic from the SPAN port to a file.

3. Solr Configuration and test methodology

Using the testbed and the performance monitoring framework described above, we ran a number of experiments to identify the relationship between the 95th percentile and the mean of the response time with the CPU and RAM allocated to the Solr VM for different values of the query arrival rate λ. The resource reserve of the ESX based virtual machine was set equal to the resource limit so that the CPU and RAM resources for a VM do not change duration an experiment. The maximum and minimum heap sizes of the Jetty webserver were set in each test. Since Jetty executes Solr as a Java servlet, increasing the memory of the virtual machine without increasing the memory of the Jetty webserver's heap would not allow the Solr process to take advantage of the RAM resources. The heap size is adjusted using the Xms and Xmx parameters which control the minimum heap size and maximum heap size respectively. To reduce overhead incurred by the Java memory manager from growing and shrinking the heap size dynamically, the Xms and Xmx parameters were set to the same value for all tests. This practice is also used by Lulu. The Java VM uses a generational garbage collector, which divides the heap into generations according to the length of time a Java object or data structure is required by the application. The frequency, type, and length of garbage collection can be significantly affected by the Xmn parameter, which sets the size of the *young generation*. Typically, the Xmn value should be between one fourth and one third of the Xmx value [6]. Before each test the Xms, Xmx, and Xmn values were set and Jetty was restarted.

An additional RAM consideration for Solr is the Operating System's ability to cache the entire index into memory. The Lulu Virtual Machine has an index size of 407 MB, and therefore the OS should at all times have at least 407 MB of free RAM to cache the index into memory. This caching is done at the disk block level and occurs transparently to Solr. Our testbed additionally monitors memory usage using the *free* command in Linux which allows us to monitor the operating system's ability to cache the Solr index [5].

Solr has three separate caches, associated with the single Index Searcher which provides a view of the index documents. The filterCache stores unordered sets of document IDs. The query result cache stores ordered sets of document IDs, and is useful for repeat queries. The document cache stores Lucene document objects that have been fetched from the disk. Solr Cache sizes are defined in terms of the number of objects the cache will contain. The Lulu Solr virtual machine has all three caches enabled, each with a cache size of 512. We did not evaluate the impact of Solr caches on the response time, and we did not modify the cache sizes from the original values Lulu set them to.

Each experiment consisted of fixing the CPU and RAM and the arrival rate λ. Then, 3000 queries were generated with a mean inter-arrival time of $1/\lambda$, and the response time for each query was measured, such that the mean and 95th percentile can be computed offline. Once the experiment was over, the virtual machine was turned off and reverted to the original virtual machine snapshot. This has the effect of reseting the Solr cache to the state the virtual machine was in before the experiment, thus allowing the same 3000 queries to be run on the next configuration without caching speedup advantages skewing the results.

We note that for various combinations of CPU, RAM and λ, Solr becomes unstable, that is, its response time becomes very large. From queueing theory, a queue is stable when its arrival rate is less than the service rate. Otherwise, the queue is unstable and the waiting time becomes infinitely large. Solr runs multiple threads, the number of which varies from 25 to 200 and the VM runs on two virtual cores. Consequently, it is difficult to obtain an analytic condition of stability, and it is also beyond the scope of this paper. In view of this, the stability region for combinations of CPU, RAM and λ were established by trial and error. To determine an unstable configuration, the mean response time was computed over a moving average of 50 observations. Figure 1 and 2 give this moving average for two different configurations. If the moving average response times at the beginning and ending of the tests are near each other, the configuration is stable since the overall response time trend has not changed considerably. This is because, when Solr is stable it alternates between busy and idle periods. If it is unstable, it is always busy, and its queues increase continuously which causes the response time to increase continuously as well. The configuration in Figure 1 is stable whereas the one used in Figure 2 is not. Failing TCP connections and the stalling of new TCP connections are also good indicators of system instability.

We note that in Figure 1, there is a spike early on. This is most likely due to I/O latency from Solr bringing large portions of the index into memory. Most Solr performance studies assume the server has reached steady state at which point the index would already be cached by the OS into memory. In this study, the OS and Solr caches are warmed with 1500 queries, and then tested with another 1500 queries for a total test length of 3000 queries. The initial 1500 queries always contain the spike shown in Figure 7.

A variety of experiments were run by varying the CPU (Mhz), RAM (MB), and the arrival rate λ, as follows: CPU = [512, 1024, 1536, 2048], RAM = [512,

1024, 1536, 2048], λ = [0.1, 0.143, 0.2, 0.333, 1.0]. Given that a test simulates 3000 customers, and interarrival times vary between 10 and 1 seconds, a typical test length can vary between 8.33 hours and 0.8333 hours respectively.

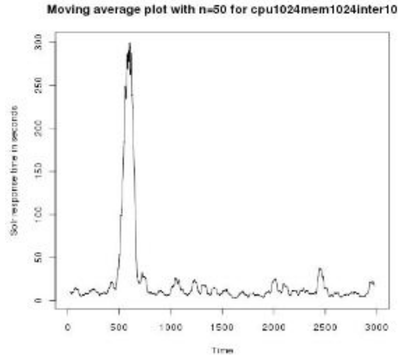

Figure 1: A stable system

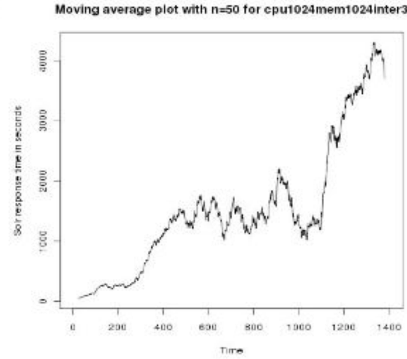

Figure 2: An unstable system

4. Results

In the first experiment, we set the CPU of the VM to 2048 Mhz and we also fixed Xmx, Xms, and Xmn to 512, 512, and 128 respectively. We measured the response time by varying the RAM allocated to the VM machine from 512 MB to 2048 MB in increments of 512 MB. The results are given in Table 1. For each value of RAM we give the 95th percentile of the response time, along with the mean response time. In the last column, we also give how many MB of the index is cached in the RAM of the operating system.

Table 1: Varying VM RAM while keeping CPU, λ, Xmx, Xms, and Xmn constant

CPU	RAM	Xmx	Xms	Xmn	95th percentile	Mean	MB of Index in RAM
2048	512	512	512	128	6.73	2.08	41
2048	1024	512	512	128	7.26	2.02	324
2048	1536	512	512	128	15.44	4.43	324
2048	2048	512	512	128	17.08	4.65	320

We observed that while CPU, λ, Xmx, Xms, and Xmn are held constant, the response time increases as the RAM of the virtual machine increases. This result is counter-intuitive considering that while the virtual machine as a whole has more memory, the application (Jetty) should not have access to any additional memory resources since the Xmx, Xms, and Xmn parameters have not been changed. Solr appears to perform poorly whenever the RAM of the virtual machine is greater than 1GB. We have not been able to explain this strange and repeatable behavior. We are inclined to speculate that it is most likely due to the specific asset Lulu has provided, and may not be repeatable across testbeds with different Solr indexes and configurations.

Note in Table 1 that response time increases as the size of index file blocks cached in RAM decreases. This is due to I/O latency introduced from Linux having to fetch data from the disk more often instead of Linux transparently keeping the index file blocks in memory. This explains why the case where RAM is 1024 outperforms the case where RAM is 512.

In a second experiment, we fixed the VM CPU and RAM to 2048 MHz and 2048 MB respectively, and varied Xmx and Xmn with Xms = Xmx. The results are presented in Table 2 including the 1st (25%), 2nd (50%), 3rd (75%), 95th percentile and mean response times. We first note that in the case of Xmx = Xms = [1792, 2048], the entire Lulu index cannot be cached in the OS, but only 155 and 34 MB, which produce 95th percentiles of 29.97 and 40.18 respectively. This is consistent with the observations made above based on Table 1.

Table 2: Varying Xmx and Xms values while holding CPU, OS RAM, Xmn, and λ constant.

CPU	RAM	Xmx	Xms	Xmn	1st quantl	2nd quantl	3rd quantl	95th perc	mean
2048	2048	256	256	128	0.82	2.72	8.9	23.74	6.24
2048	2048	512	512	128	0.79	1.96	6.68	16.99	4.49
2048	2048	768	768	128	0.82	2.07	6.93	17.31	4.84
2048	2048	1024	1024	128	0.8	1.95	6.7	17.56	4.72
2048	2048	1280	1280	128	0.78	2	6.63	17.95	4.83
2048	2048	1536	1536	128	0.81	2.25	7.11	18.5	5.19
2048	2048	1792	1792	128	0.93	2.61	7.53	24.43	6.55
2048	2048	2048	2048	128	1.2	2.84	8.48	29.95	7.65

Table 3: Response time in seconds with λ=0.1

CPU (MHz)	RAM (MB)	Average (s)	50th Percentile	95th Percentile
2048	512	1.9	1.14	5.89
1920	512	2.12	1.17	6.81
1792	512	2.56	1.25	8.75
1664	512	3.01	1.35	10.65
1536	512	3.17	1.34	12.18
1408	512	3.76	1.5	14.42
1280	512	5.29	1.97	20.73
1152	512	5.86	3.03	21.3
1024	512	10.35	4.84	39.29

Table 4: Response time in seconds with λ=0.111

CPU (MHz)	RAM (MB)	Average (s)	50th Percentile	95th Percentile
2048	512	2.36	1.26	8.19
1920	512	2.3	1.27	7.89
1792	512	2.73	1.36	9.49
1664	512	3.3	1.43	11.8
1536	512	3.33	1.49	11.66
1408	512	4.85	2	19.22
1280	512	6.06	2.24	23.66
1152	512	7.52	4.05	27.99
1024	512	12.74	7.58	48.01

Based on the above observations, we carried out a third experiment where we varied the CPU. The VM's RAM was fixed to 512 because the entire index was able to be fit into the cache, and the Solr caches were fixed so that they have their full-effect, i.e. Xmx = Xms = 512 and Xmx = 128. CPU (Mhz) and the arrival process (λ) were varied as follows: CPU = [1024, 1280, 1536, 1792, 2048] and λ = [0.1, 0.111, 0.125, 0.143]. The data obtained are given in Tables 3, 4, 5 and 6.

Table 5: Response time in seconds with λ=0.125

CPU (MHz)	RAM (MB)	Average	50th Percentile	95th Percentile
2048	512	2.23	1.26	7.28
1920	512	2.72	1.35	9.9
1792	512	3.12	1.39	12.02
1664	512	3.44	1.35	13.3
1536	512	4.3	1.81	14.8
1408	512	5.53	2.48	20.43
1280	512	6.26	3.26	23.71
1152	512	9.14	4.34	32.82
1024	512	14.89	7.86	55.94

Table 6: Response time in seconds with λ=0.143

CPU (MHz)	RAM (MB)	Average (s)	50th Percentile	95th Percentile
2048	512	2.65	1.38	9.25
1920	512	2.71	1.25	9.7
1792	512	3.8	1.59	15.59
1664	512	4.46	1.86	17.36
1536	512	4.32	1.83	15.59
1408	512	6.03	2.7	23.89
1280	512	8.18	4.09	30.16
1152	512	13.12	7.08	52.2
1024	512	26.21	11.85	98.74

We only present results for λ ≤ 0.143 because larger values of λ result in the Solr system to become unstable as described in section 4.2 and shown in figure 2. Notice that in all stable cases presented, the inter-arrival time (1/λ) exceeds the average service time. This is acceptable since as mentioned above, Solr uses multiple threads. Projecting out the trend of increasing the response time as λ increases, a rate of one customer every 6 seconds, or λ = 0.1667 would have reached the crossover point where the average response time for most CPU values is greater than 6 seconds causing an unstable system.

Table 7: Coefficients and R^2 values for the 95th percentile of the response time

λ	a_0	a_1	R^2
0.143	102.3332	-13.1166	0.9641
0.125	58.7735	-6.8516	0.9449
0.111	50.7081	-5.8344	0.9227
0.100	41.2238	-4.6720	0.9316

Based on the results given in Tables 3 to 6, we fitted the 95th percentile and the mean of the Solr response time R (sec) to VM's CPU (Mhz), denoted as C_{mhz}, using the expression $R = a_0 + a_1 \log(C_{Mhz} - 1023)$. Tables 7 and 8 give the coefficients a_0 and a_1 and the R^2 values for various values of λ for the 95th percentile and the mean of the response time respectively.

Table 8: Coefficients and R^2 values for the mean response time

λ	a_0	a_1	R^2
0.143	27.0087	-3.4708	0.9752
0.125	15.7023	-1.8140	0.9366
0.111	13.3846	-1.5227	0.9382
0.100	10.8054	-1.1981	0.9464

4. Conclusions

In this paper we presented an analysis of the relationship between the mean and the 95th percentile response time of the Lulu search engine with the CPU and RAM resources allocated to the VM of the search server and the Solr caches as a function of the arrival rate. The stable, sustainable traffic arrival rates have been identified to be less than λ = 0.143 for a single virtualized Solr server with CPU ≤ 2048. Given an index size of 407 MB and all Solr caches configured to 512 entries, 512 MB of VM RAM produces optimal performance. The required CPU for a given arrival rate of querying can be determined using an expression that we obtained based on observed data.

5. References

[1] Lulu. www.lulu.com

[2] D.F. Garcia, J. Garcia, M. Garcia, I. Peteira, R. Garcia, and P. Valledor. Benchmarking of web services platforms: An evaluation with the TPC-App benchmark. In: Proc 2nd Int. Conf. On Web Information Systems and Technologies, p 75-80 (2006).

[3] Wireshark. www.wireshark.org

[4] Snort: www.snort.org

[5] free. http://linux.die.net/man/1/free.

[6] Scaling Lucene and Solr. http://www.lucidimagination.com/Community/Hear-from-the-Experts/Articles/Scaling-Lucene-and-Solr

[7] Tsung. tsung.erlang-projects.org.