WISE 2.0: An Improved Toolkit that Now Supports the Noisy Neighbor Experiment

Final Report by Raghav Bhat

Spring 2020, Intro to Enterprise Computing

Tool Location: https://github.com/rbhat35/WISETutorial

Introduction

This report highlights my contributions to the next-generation WISE Toolkit.[1] The WISE Toolkit has been developed to aid in finding performance bugs caused by millibottlenecks.[2] In my project, I have advanced this goal in the following ways.

- Adding Support for the Noisy Neighbor Experiment
- Automating Experimental Procedure
- Key QA Contributions
- Fixing the Memory Profiler

These contributions are critical in supporting future research in the field of performance research, as they add novelty, scalability, and credibility to the experiments that can be run on the benchmark.

Note: Readers already familiar with the Millibottleneck Theory of Performance Bugs and the WISE Toolkit may skip the next two sections and go to the section titled Contributions to WISE Toolkit.

Background

For many web servers, the vast majority of requests take 10s of milliseconds [1]. However, when response times are plotted, one will notice that a small subset of requests take significantly longer—on the order of seconds [1]. As it turns out, the distribution of number of requests vs. response time has a long-tail [e.g. Figure 1]—a very high concentration of requests is returned very quickly, but a few requests take slightly longer, and far fewer take substantially longer (i.e. they rapidly taper off). Note that the longer requests are not inherently slower (e.g. more complex queries)—the response time is an artifact of the system, even at low resource utilization rates. Although it may come as no surprise that some requests take longer than others, what is, perhaps,

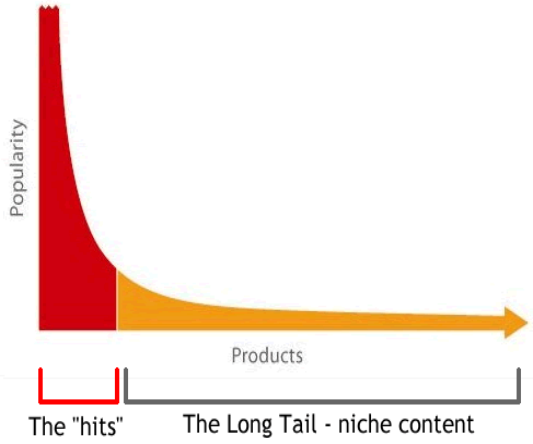

Figure 1

[1] See WISE Toolkit & Benchmark
[2] See Background

surprising in this case is that the difference in response time can be several orders of magnitude.

Due to the *very* long nature of the tail, it is problematic. According to an Amazon study, a 1% loss in sales was tied to every 100 milliseconds increase in page load time [1]. Google found a similar result, noting that if search took 500 milliseconds longer, the company stood to lose up to 20% of its revenue [1]. Thus, there is a strong financial incentive to tackle the latency long tail problem, and yet, it is not on the vast majority of developers' minds. This is likely because we know little about what causes these latency long tails and how to solve them.

According to Pu et. al, the long tail is caused by "millibottlenecks" [1]. The idea is that a milliseconds-long bottleneck in one part of an n-tier system will propagate through the entire system. Thus, this split-second bottleneck's effects are enlarged and can cause queues in other systems to overflow, for example [1]. This theory has been substantiated by running a standard benchmark (RUBBoS) and logging response times, as well as resource utilizations, at the millisecond level [1]. After correlating different events, the seminal paper [1] shows that the JVM garbage collector can cause millibottlenecks.

However, there are several challenges in detecting and triaging these millibottlenecks. Firstly, resource monitoring must be very fine-grained, capable of detecting millibottlenecks of just 100 milliseconds. Most resource monitoring tools work at a far coarser granularity (seconds) and thus won't help detect and correlate millibottlenecks due to the sampling theorem. Secondly, the monitoring tools need to have very low overhead, so as to not introduce noise. This is especially true since we might need to simultaneously monitor several resources (e.g. CPU, memory, queue sizes). Finally, finding resource millibottlenecks simply illustrates their effect; we must log the system events that are their cause, with the hope of correlating cause and effect.

One way to correlate, and therefore better understand, millibottlenecks with their cause is to run system experiments. System experiments will allow us to observe several millibottlenecks and gather the data to determine why they are happening. Eventually, this knowledge can be helpful in beginning to solve them. An experimental, as opposed to theoretical, analysis approach is best suited since we are only beginning to understand the effects of millibottlenecks and hardly have a grasp of the dependencies that might cause them.

WISE Toolkit & Benchmark

We now detail the framework that has been developed by Professor Calton Pu's lab at the Georgia Institute of Technology to run experiments in search of millibottlenecks. Specifically, we discuss the toolkit and the benchmark (workload). The experiments are designed to be portable, so we can test the effects of different hardware and software.

WISE Toolkit

There are many challenges related to studying millibottlenecks. Firstly, the scientific process requires that the experiments are reproducible. Since millibottlenecks are system-level, all the dependencies the system has *must* be captured—for example, the specific hardware configuration and cache sizes must be known [5]. Secondly, there is the issue of data granularity. Most system monitoring tools monitor at the second or minute level. As discussed in the previous section, this is not sufficient for us. Finally, we must be able to reconstruct the execution path of requests in a distributed environment. This means we must have precise logging of events on every node in our system [5].

In order to meet these requirements, researchers at Georgia Tech have developed the Elba toolkit. The resource and event monitoring component of Elba is called milliScope. The resource monitoring component of milliScope uses

open source tools such as *sar*, *Collectl*, and *iostat* to monitor the CPU, memory, network, disk, etc. at a fine granularity, with low overhead [3]. On the other hand, the event monitoring component of milliScope specializes the software components, retooling the native log components to log the execution boundaries of events (i.e. when the response goes in and out of the component).

In order to meet dependency requirements, Elba uses a workflow language called WED-Make to captures explicit and implicit dependencies [5]. With WED-Make, one can specify a complex benchmark, as is illustrated in paper [3].

WISE Benchmark

In order to conduct experiments, we must have a workload to run. The second-generation Elba toolkit, code-named WISE, includes a twitter-like web application that we use. Simulated tasks include creating and endorsing posts, subscribing to other users, viewing posts, and viewing a user's own inbox [6]. Following current industry trends in architecture, the application is microservices-based. As we can see in Figure 2, which details the architecture, the client sends an HTTP request to an Apache server, which then relays requests to the appropriate Thrift microservice [6].

Each of the five microservices connected to the Apache server handles a different aspect of the workload. Auth handles user registration and authentication, and sub manages user subscriptions; microblog provides core functions, such as creating posts, while queue is used to offload tasks that can be handled asynchronously; inbox allows the user to push and pull to his or her inbox. Stateful data is stored in a PostgreSQL database, and there is a worker node that asynchronously processes requests relegated to the queue. [6]

The workload itself, which consists of one or more clients simulating user requests, is highly configurable. Configuration parameters include how many clients are sending requests and for how long [6].

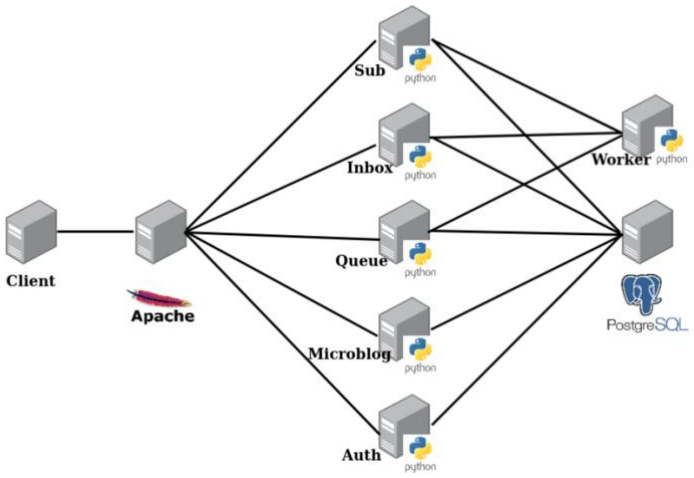

Figure 2 [6]

CloudLab

The WISE Toolkit is designed to run on CloudLab [4]. CloudLab is a public cloud for researchers and scientists, allowing researchers to provision geographically dispersed machines with several different kinds of hardware and software.

Contributions to WISE Toolkit

In this section, I highlight my improvements to the WISE Toolkit, why they were necessary, and how they help future researchers.

Adding Support for the Noisy Neighbor Experiment

The noisy-neighbor phenomenon occurs when different virtual machines on the same host interfere with each other—for example, if another virtual machine on the same host suddenly has a burst in workload, the performance of your virtual machine will be impacted [7]. The noisy-neighbor problem was first published in [7] using an older version of the Elba toolkit and a different benchmark, RUBBoS[3]. However, this generation's new, microservice-based benchmark did not support the noisy neighbor experiment out of the box.

One way to simulate a noisy neighbor is to use a stress testing tool and run the tool inside a virtual machine. By running the virtual machine in the same host as one or more of our services, we can compare how our system behaves with and without the noisy neighbor. For example, running a disk-heavy stress test on the same host machine as the database in our benchmark will very likely slow the rate of database transactions, which will have a cascading effect on the performance of other services, thus leading to very long response times.

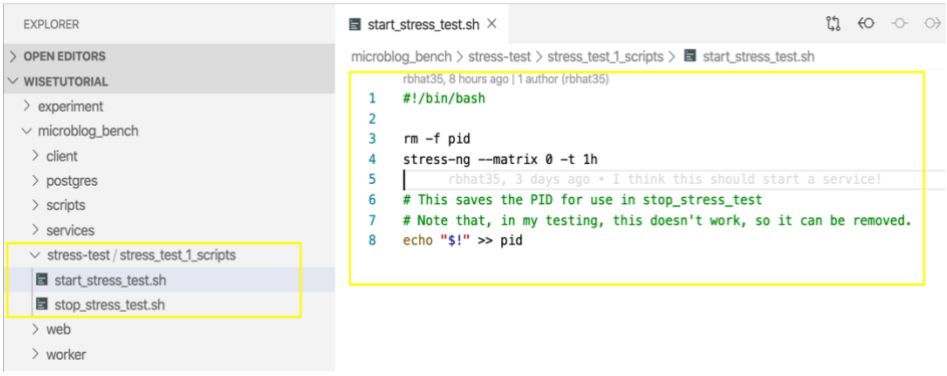

Figure 3

In order to add support for this experiment, I instrumented WISE so that during benchmark execution, a stress test can also be run. The experimenter simply specifies the hostname(s) of the virtual machine(s) the stress test should be run on in the config.sh file. Additionally, the stress test can be easily configured by the researcher. By simply editing the script depicted in Figure 3, he or she can run any stress test that stress-ng supports. Thus, configuration required to simulate a noisy neighbor is extremely minimal, yet very powerful.

Without this feature, a researcher seeking to run the noisy neighbor would have had to run the benchmark manually by using ssh to access a node in the OpenStack cluster. Not only does my feature make this experiment far easier to run and more scalable, but we also continue to run all the monitor tools on the VM running the stress test. Hence, we get useful log data that validates whether our stress test ran successfully.

Automating Experimental Procedure

As discussed, reproducibility is a key tenet of experiment-based research. In order to aid in reproducibility, I fully automated the testing pipeline on OpenStack & bare metal. Thus, I increased both the efficiency and scale at which future experiments can be run while also reducing the friction required to reproduce experiments.

The experimental workflow has been vastly simplified. Figure 4 and Figure 5 depict the experimental procedure before and after automation was completed, respectively. In Figure 5, we can see that the experimental procedure

[3] The RUBBoS application benchmark is a four-tier system that simulates user requests to a Reddit-like website [3].

has been fully automated. There remain, of course, some manual steps that are unavoidable, such as configuring the parameters of the experiment. However, all the repetitive tasks—such as parsing the results—that were previously manual processes have been automated.

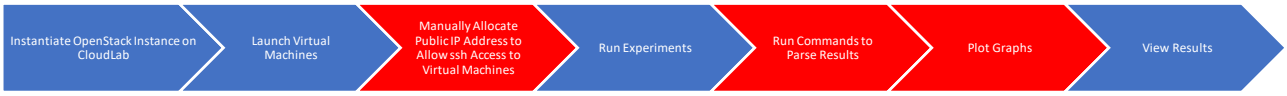

Figure 4

Figure 5

Automatic IP Allocation

One of the major barriers in fully automating the pipeline was the need to assign a public-facing IP address to one of the nodes, which is necessary to allow ssh access. Previously, this was done manually through the OpenStack dashboard. However, using the OpenStack Python API, I automated this step so that a public-facing IP address is allocated in the same script that deploys the virtual machines.

This step vastly increases the productivity of a researcher for a number of reasons. First and foremost, the WISE toolkit currently has a limitation in that experiments can only be run once on a given cluster of virtual machines. In other words, between every two experiments, one must destroy and recreate all the virtual machines—and hence, must also allocate a public-facing IP address. It's important to note that a researcher might run 30+ experiments for a single project, and hence even a simple repetitive task can easily become a nuisance. Secondly, the OpenStack dashboard on CloudLab is, at times, slow and unresponsive. Thus, I was able to eliminate one of the most frustrating steps in the whole experimental procedure.

Parsing & Plotting Scripts

Once experiments finish execution, the raw results have to be parsed into CSV files that can then be plotted. The toolkit already provides all the necessary parsing scripts; however, the researcher is expected to call each parser from the command line to generate the data he or she is interested in. This amounted to, essentially, copying and pasting a series of commands from the tutorial to the shell. Once the relevant data has been plotted, the researcher can then run the appropriate gnuplot script.

In order to speed up the process of graph generation, I created a shell script to parse and automatically plot *all* the data. After running ./experiment/parse_results.sh, all the plots are neatly saved in a new plots/ directory, as can be seen in Figure 6. The data CSVs are structured in a similar format as well. Thus, once the experiment completes, the researcher can move straight to the analysis phase—saving a computer scientist's most valuable resource, time.

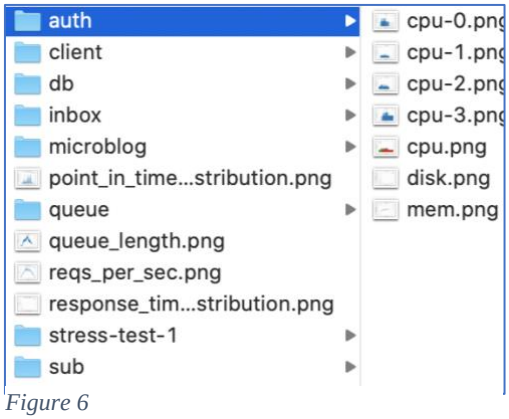

Figure 6

Fixing the Memory Profiler

As discussed, the Elba toolkit provides resource utilization level at an extremely fine-grained level. This data can be used to spot very long response times, as well as diagnose why they might have occurred. Thus, it is paramount that the toolkit provides complete and accurate results.

After running one set of experiments, I noticed that the memory plots appeared to show no memory usage—the graphs were a horizontal line, parallel with the x-axis with memory utilization at a constant 0%.

Of course, this could be for one of two reasons—either the memory log collector in WISE was not working correctly or the parsers were not parsing the data properly. I noticed that the raw memory logs in the experimental results were non-empty, and thus, it was possible that the issue lay in the parser.

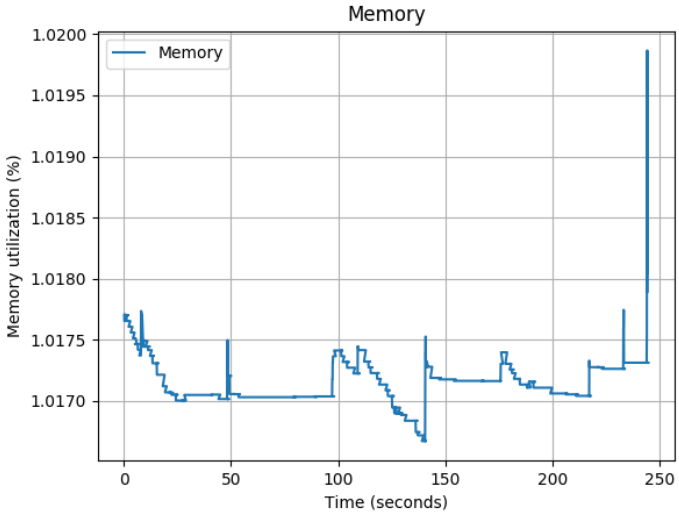

Figure 7

When walking through the memory parser file with the debugger, I noticed that we were rounding numbers quite early in our calculations. After increasing the precision of the floating points we were using, a sensible plot was visible. The flat-line plot turned into Figure 7.

However, as you can see in Figure 7, while the graph shows a reasonable trend, the percentage of memory actually used is very low, hovering around 1% of total system memory. This is obviously concerning and suggested to me that perhaps the WISE memory monitoring tool was flawed, since even kernel activity will easily consume more than 1% of memory on virtually any system.

I began my investigation by looking at the raw memory logs generated by our monitoring toolkit. Interestingly, these logs reported total system memory as 265GB. After reading the CloudLab documentation [4] and using the top Linux utility to monitor memory, I found the following figures for how much RAM is available at different system abstractions.

System	Memory (RAM) Allocated
Virtual Machine running on OpenStack Node	8GB
OpenStack Node	26GB
Clemson c8220 Cluster	256GB

It became clear that our auditing tool was somehow listing the total system memory as the memory available on the entire cluster. In other words, the 265GB of total system memory should have actually been 8GB in the report. While this is certainly less than ideal, we are typically more concerned about the change in memory usage over time rather than the absolutely quantity of memory used itself. Thus, even though the y-axis in Figure 7 is off by a constant factor, the trend is still correct.

It is also important to note that, on average, memory utilization while the benchmark ran hovered around 2.4GB per virtual machine. Thus, given that the toolkit believed the total system memory was 265GB, our memory plots with ~1% utilization actually make sense.

Key QA Contributions

Overall, I also helped debug and test the initial version of the tutorial and toolkit published this semester. I was one of the first students in the class to run an experiment, and thus I helped find bugs that other students were able to avoid.

Bugs I found & Helped Fix

1. Discrepancy in WISE Tutorial—When running experiments on bare metal hosts, one must change the default shell on CloudLab from tcsh to bash, which is the shell most of the scripts we run use [4]. This was originally not in the tutorial.
2. Critical Files were Missing Files in repository—In the first few commits of the WISE Toolkit, a few critical files were accidently being ignored, causing the experiments to fail. I brought this issue to Rodrigo's attention, and he promptly fixed it.

Challenges Faced

Over the course of my project, its scope and deliverables have changed substantially. I had originally set out to run several experiments that would explore the noisy neighbor phenomenon. Instead, I ended up focusing on modifying the toolkit so that doing the set of experiments I had proposed would be far easier for future researchers. Although coronavirus helped necessitate this change, the main reason my project ended up diverging substantially from my proposal is because the tool had many limitations that made running noisy neighbor experiments at scale infeasible. These were limitations I was not aware of at the time I proposed my project, and I believe I have eliminated most of these limitations over the course of the semester.

Ultimately, the key limitation that we all face is a lack of time. Before my changes, running the 20-30 experiments I had proposed would easily have taken twice as long as they would now. When I realized how long these experiments would take to run, I considered pivoting. Secondly, the bugs I encountered as I went through the tutorial were often blocking issues. Thus, I spent several days at the beginning of the project working on bug fixes. I did not anticipate so many issues at the time of proposal, and since most of these bug fixes were necessary to get the tool working, ignoring these issues was not possible.

Finally, although I was aware that the WISE Toolkit did not support the noisy neighbor experiment out of the box, I did not understand how difficult adding this feature would turn out to be. Since the experimental setup happens in a single bash script with nearly a thousand lines, it was extremely difficult to determine what changes were necessary to run my stress test. Additionally, once I made a change, I had to test it—meaning I would have to run the script, which takes over an hour and requires some manual configuration. Thus, adding support for the noisy neighbor experiment alone took nearly a week, which is time that was previously allocated for running experiments.

Thus, while I did not necessarily achieve what I had set out to, I still believe I accomplished a substantial body of work that, on the face of it, seemed like it would be far easier.

Conclusion

Overall, I believe the contributions I have made to the WISE Toolkit can substantially help future research in the field of performance testing and profiling. I helped with several bug fixes to the WISE toolkit and WISE tutorial,

helping not only my classmates this semester, but also for semesters to come. My contributions in experimental automation will allow researchers to collect data at scale, and for a class of experiments—the noisy neighbor experiments—that were not possible before. While I ended up delivering something different from what I had originally set out to do, in many ways, what I have been able to contribute is far more significant.

In retrospect, I have realized that this project taught me a lot. Firstly, projects rarely go as planned—and it's important to be able to pivot and find success. Secondly, I have gained new appreciation for tools and language features I had previously taken for granted, as well as software development techniques that I was forced to adopt as a result of this projects difficulties. Finally, technical decisions made early on can have a significant impact on a product's outlook—a monolithic bash file with hundreds of lines is quite difficult to work with and thus ended up causing a feature that should have taken 1-2 hours to take 10+ hours of research and development.

References

[1] The Millibottleneck Theory of Performance Bugs, and Its Experimental Verification (Calton Pu et al.)
[2] The Tail at Scale (Jeffrey Dean and Luiz André Barroso)
[3] Systematic Construction, Execution, and Reproduction of Complex Performance Benchmarks (Rodrigo A. Lima et al.)
[4] CloudLab, https://www.cloudlab.us/.
[5] Elba Std Project Slides, by Rodrigo A. Lima. Presented in class Tuesday, February 11, 2020.
[6] Elba Project Tutorial 2, by Rodrigo A. Lima. https://www.cc.gatech.edu/~ral3/tutorial2.html
[7] Q. Wang, Y. Kanemasa, J. Li, C. Lai, C. Cho, Y. Nomura, and C. Pu, "Lightning in the cloud: A study of transient bottlenecks on n-tier web application performance," in *2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA*, 2014.
[8] OpenStack, https://www.openstack.org.
[9] stress-ng, https://wiki.ubuntu.com/Kernel/Reference/stress-ng.