Limitations of Load Balancing Mechanisms for N-Tier Systems in the Presence of Millibottlenecks

Tao Zhu[1], Jack Li[1], Josh Kimball[1], Junhee Park[1], Chien-An Lai[1], Calton Pu[1] and Qingyang Wang[2]
[1]*Computer Science, Georgia Institute of Technology* [2]*Computer Science and Engineering, Louisiana State University*

***Abstract*—The scalability of n-tier systems relies on effective load balancing to distribute load among the servers of the same tier. We found that load balancing mechanisms (and some policies) in servers used in typical n-tier systems (e.g., Apache and Tomcat) have issues of instability when very long response time (VLRT) requests appear due to millibottlenecks, very short bottlenecks that last only tens to hundreds of milliseconds. Experiments with standard n-tier benchmarks show that during millibottlenecks, some load balancing policy/mechanism combinations make the mistake of sending new requests to the node(s) suffering from millibottlenecks, instead of the idle nodes as load balancers are supposed to do. Several of these mistakes are due to the implicit assumptions made by load balancing policies and mechanisms on the stability of system state. Our study shows that appropriate remedies at policy and mechanism levels can avoid these mistakes during millibottlenecks and remove the VLRT requests, thus improving the average response time by a factor of 12.**

I. Introduction

N-tier systems are one the best examples of scalable distributed systems that provide QoS (quality of service, e.g., guaranteed response time) for production web-facing applications such as e-commerce. The scalability of n-tier systems is largely due to: (1) their ability to allocate servers dynamically according to load, and (2) effective load balancers to spread requests among the available servers in each tier to minimize response time and maximize throughput. For web-facing applications, n-tier systems have been able to provide good response times, as fast as milliseconds. However, they are also known for the lingering problem of long-tail distribution of response times, where some very long response time (VLRT) requests take several seconds to return results.

The VLRT requests are a serious and perplexing problem for web-facing applications. They are serious because deviation from strict QoS is bad for business: A study by Amazon [16] reported that every increase in response time of 100 milliseconds is correlated to roughly 1% loss in sales. Similarly, Google found that a 500ms additional delay in returning search results could reduce revenues by up to 20% [17]. Given the number of customers they serve, companies such as Amazon and Google want to reduce the response time long tails to the 99th and 99.9th percentiles [12], [13]. The VLRT requests are also a perplexing problem since they appear with modest utilization levels, e.g., 50% of CPU utilization.

One of the causes of VLRT requests consists of millibottlenecks [27], [26], [28] that appear and disappear within tens to hundreds of milliseconds. Millibottlenecks appear quite commonly for a variety of reasons, including bursty workloads and system tasks such as garbage collection [12], [27], [26], [28], [19]. The resources that suffer from millibottlenecks include CPU [27], [26], [28] and I/O [19]. Millibottlenecks can cause VLRT requests for several reasons, typically dropped packets due to Cross-Tier Queue Overflow. (Section III includes a more detailed discussion of millibottlenecks and their effects.)

Experimental measurements on standard n-tier benchmarks have found load balancing policies and mechanisms that appeared to work well in stable environments have exhibited several limitations when facing millibottlenecks. For example, load balancing policies that use cumulative requests served may send requests to a server that did less work historically, but that is suffering from (sudden) millibottlenecks at this moment. The first contribution of the paper consists of experimental evidence showing several such limitations of current load balancing policies and mechanisms in the presence of millibottlenecks.

The second contribution of the paper consists of proposed changes to load balancing policies and mechanisms that take into account the millibottlenecks, e.g., adding the consideration of recent utilization changes. Measurements show that the impact of millibottlenecks on n-tier system response time can be reduced by a factor of up to 12, when revised load balancing policies and mechanisms are applied.

The rest of the paper is structured as follows. Section II describes current Apache load balancing policies and demonstrates their effectiveness in an environment without millibottlenecks. Section III shows millibottlenecks interfere load balancer. Section IV analyzes the load balancer instability caused by implementation limitations and provides remedies to remove the limitations. Section V analyzes the instability caused by limitations at policy level and demonstrate how our remedies can avoid the issue. Section VI summarizes the remedies at both implementation and policy levels. Section VII summarizes the related work and Section VIII concludes the paper.

II. Current Apache Load Balancers

A. Background on Current Apache Load Balancing

N-tier is a classic architecture of modern web application, which is usually implemented as three or more tiers. A typical 3-tier configuration uses Apache HTTP server as web server, Tomcat as application server and MySQL as database

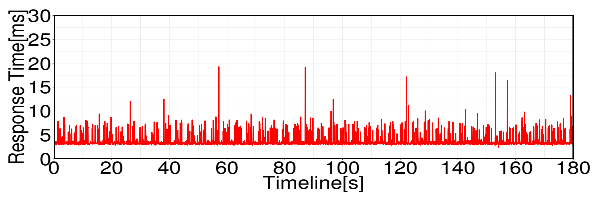


Fig. 1: Point-in-time response time under the total_request policy

server. Our experiments run the standard 3-tier. RUBBoS benchmark [2], whose workload consists of 24 different web interactions modeled after the typical bulletin board interactions with Slashdot. In this paper, we study primarily the load balancers that Apache web server uses when choosing an appropriate Tomcat application server. Apache tomcat connector (mod_jk) [3] is the module that implements (two) load balancing policies:

1) Total_request (default policy): ranks the Tomcat servers according to their accumulated number of requests served (fewest as the best candidate).
2) Total_traffic: ranks the Tomcat servers according to the accumulated messages exchanged between Apache and Tomcat (fewest as the best candidate).

These policies are implemented by a two-level scheduler. The higher level (one for each policy) calculates the ranking of each server, according to each load balancing policy. For example, the number of requests (policy 1) and number of messages (policy 2) are translated into a normalized value, called lb_value. The lower level (same for all policies) uses the lb_value to schedule the next request.

B. *Validation of Apache Load Balancers*

The RUBBoS benchmark includes two kinds of workloads: browsing-only and read/write interaction mixes. We ran RUBBoS workload at 70000 clients using 4 Apache servers, 4 Tomcat servers, and 1 MySQL server. To see how load balancers work in the absence of millibottlenecks, we adopted remedies to eliminate all known millibottlenecks. For instance, to eliminate the millibottlenecks caused by flushing dirty page, we enlarged the memory that holds the dirty pages and lengthened flushing interval.

We ran baseline experiments to confirm that the Apache load balancers work well in the absence of millibottlenecks. The total_request policy achieved good performance: average response time is 3.2ms. The total number of VLRT (>1 second) request during the experiment duration is 13, which is negligible considering the total volume of requests is more than 1.8 million. The stable and low point-in time response time in Figure 1 confirms that millibottlenecks are eliminated and suggests the load balancer works as it is supposed to.

To verify the effectiveness of the total_request policy, we analyzed both the Apache and Tomcat logs to see how each Apache server distributed workloads among the Tomcat servers during the experiment. We found Apache server distributed the workload evenly among the Tomcat servers.

III. Millibottlenecks Interfere With Load Balancers

A. *Millibottlenecks and VLRT Requests*

For completeness, we include a short introduction to millibottlenecks here. Readers who are familiar with millibottlenecks, previously called very short bottlenecks [28] and transient bottlenecks [26], can skip to the next section III-B.

Past studies have examined and delivered valuable insights of causes on the VLRT requests. VLRT requests can have very different causes, traversing the system stack. These include CPU dynamic voltage and frequency scaling (DVFS) control at the architecture layer, Java garbage collection (GC) at the system software layer, virtual machine (VM) consolidation at the VM layer and bursty workloads [27], [26], [28]. The millibottlenecks in our experiments are caused by flushing dirty pages.

B. *Millibottlenecks Cause VLRT Request in a Cluster without Load Balancer*

In this section, we ran experiments with the simplest configuration (1 Apache server, 1 Tomcat server and 1 MySQL server) in an environment where millibottlenecks occur.

First, we find the VLRT requests in an experiment. Figure 2(a) shows an 8-second time interval in which two clusters of VLRT requests can be seen (at approximately 52.5 and 53.5 seconds from the beginning of the experiment). The VLRT requests contribute the long-tail response time, the total number of requests whose response time is more than 1000ms is 1222 while the total number requests whose response time is less than 10ms is 16722.

Second, Figure 2(b) shows queued requests in Apache, Tomcat and MySQL during this same time interval. We use queue length graph to determine if there are millibottlenecks: large spikes in the graph represent an abnormally large number of queued requests, which from our experience are usually indicative of bottlenecks. On the other hand, the steady short queue length of a tier (or server) might indicate the tier (or server) is performing well. We see a strong correlation between peaks of Apache's queue length in Figure 2(b) and peaks in VLRT requests in Figure 2(a). A detailed analysis of the Apache queues (omitted here) [28] shows that packets are being dropped by Apache due to queue overflow. VLRT requests materialize due to dropped packets being retransmitted.

To find out which server has the millibottleneck, we conduct per-server queue analysis by integrating and comparing the requests queued in Apache with the requests queued in Tomcat. We establish the link between the queuing in Apache with the queuing in downstream servers by comparing the queue lengths of Apache, Tomcat, and MySQL in Figure 2(b). The first Apache queue peak indicates the existence of the millibottleneck in Apache. The second Apache queue peak coincides with the peak of Tomcat queue and there is no queue peak in Mysql tier. A plausible hypothesis is queue amplification that starts in Tomcat tier, propagating to Apache tier [27]. As Tomcat tier reaches full queue, a push-back wave starts to fill Apache tier's queue. When Apache tier's queue becomes full, dropped request messages create VLRT requests.

Further analysis of the queues and resource utilization of Apache, Tomcat, and MySQL shows that the flushing of dirty virtual memory pages to the disk causes iowait saturation, which in turn causes the millibottlenecks that lead to the observed elongated queues. The detailed analysis is omitted here since our focus is on the effect of load balancing in this context.

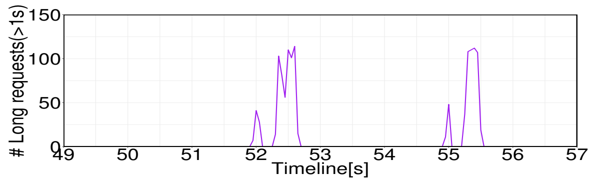
Lastly, we found pdflush contributed 100% iowait by using Iotop [4]. Pdflush is a process who is responsible for writing back dirty pages in page cache to disk. These dirty pages mainly are Tomcat logs, which include access, servelet and localhost logs. We show the sum of dirty pages change in Figure 2(e), We see the abrupt drops correlate strongly with iowait saturations in Figure 2(d), which confirms flushing dirty pages lead to transient CPU saturations in Figure 2(c). This is an unexpected result: flushing dirty pages should have minimal impact on foreground tasks since it is supposed to be asynchronous. More detailed explanation about the root cause of the millibottleneck will be discussed in another paper.

C. Load Balancers Contribute to VLRT in the Presence of Millibottlenecks

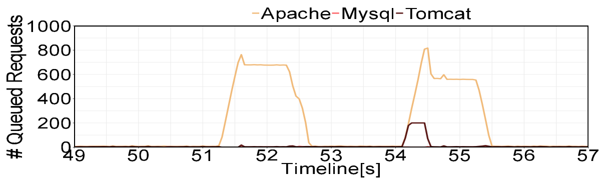
In Section II-B, we reviewed the experiments in an environment where millibottlenecks are absent. In this section, we repeated the same experiments in an environment where millibottlenecks exist. Our experimental results demonstrate the load balancing instability. The load balancer sends all of the requests to the candidate already suffering from millibottlenecks, which amplifies the magnitude of VLRT request. The load balancing instability was not revealed in the absence of millibottlenecks.

We ran the RUBBoS benchmark under two load balancing policies (total_request and total_traffic policy) at 70000 clients. Our experiments use 4 Apache servers, 4 Tomcat servers, and 1 MySQL server (The Experimental Setup details appear in the Appendix A). When looking at statistical average metrics such as response time, the performance under total_request and total_traffic policy are acceptable: their average response times are below 100ms. However, response time under the two policies present large fluctuations, as shown in Figure 3. The average system response times are not representative of the actual system performance.

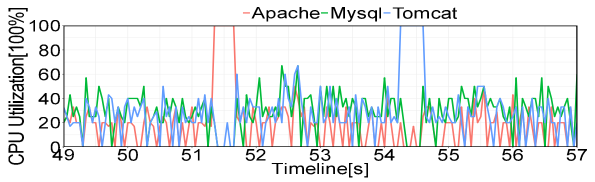
The latency long-tail problem happens when very long response time (VLRT) requests arise. Under the total_request policy, 89 percent of requests finish in 10ms, VLRT (>1000ms) requests account for 5 percent of total requests. This means VRLT requests are responsible for the tremendous increase in average response time. Total_traffic policy has a similar latency long-tail problem. By plotting the distribution of request response time, latency long-tail problem can be seen more clearly. As shown in Figure 4, we observe 3 clusters of VLRT requests (at 1s, 2s and 3s). Our result is consistent with previous research, which shows VLRT requests occur even when all system components are far from saturation. As shown



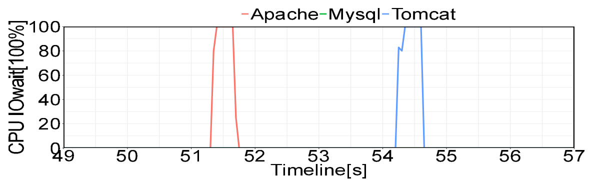
(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.



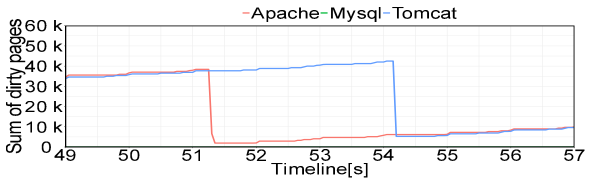
(b) Request queue for Apache, Tomcat and MySQL tier, the queue peaks match well with the occurrence of the VLRT requests. The first queue peak in Apache is caused by a millibottleneck on Apache. The second queue peak in Apache coincides with the queue peaks in Tomcat, suggesting push-back wave from Tomcat to Apache.



(c) Transient CPU saturations of Apache servers correlates with the queue spikes in the corresponding Apache server.



(d) Iowait saturations correlates with Transient CPU saturations, indicating I/O activities cause CPU saturations.



(e) Abrupt dirty page cache size drops correlate with iowait saturations, suggesting flushing dirty pages cause iowait saturations.

Fig. 2: VLRT requests caused by flushing dirty pages

in Figure 5, all the component servers were at moderately low CPU utilization (the highest average CPU usage among the servers is 45%).

Interestingly (and somewhat surprisingly), millibottlenecks in the Tomcat tier cause more severe performance degradation compared to the case without the load balancer. As shown in Figure 6(a), the total_request policy introduced more VLRT requests compared to the experiment without the load balancer. To eliminate the interference from milibottlenecks in Apache, we increased the memory that holds the dirty pages to 4.8 GB and lengthened the flushing interval to 600 seconds to ensure no millibottleneck happens on Apache during the

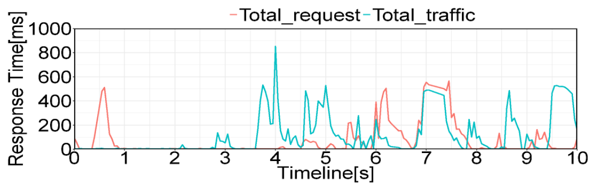


Fig. 3: Point-in-time response time of the total_request and total_traffic policies during the first 10 seconds of the experiment.

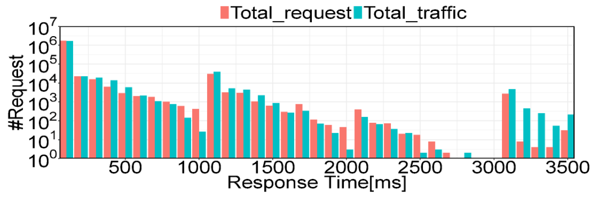


Fig. 4: Frequency of requests by their response times under the total_request and total_traffic policies.

experiment runtime. The VLRT requests are initially caused by millibottlenecks. We zoom into a period during which Tomcat experiences a millibottleneck and analyze the server-specific queues to identify the specific server experiencing the millibottleneck. We then link the VLRT requests to millibottlenecks with a fine-grained CPU utilization plot of each Tomcat server, as shown in Figure 6(b). A careful comparison of Figure 6(b) and Figure 6(a) reveals a very short period when Tomcat1 reaches full (100%) utilization coinciding with the VLRT requests. Due to space constraints, we exclude the details of diagnosing the millibottleneck. The cause of the millibottleneck is the same as the one we discussed in section III-B flushing of dirty pages on the Tomcat servers.

VLRT requests are amplified by the load balancer instability: requests are sent to the Tomcat suffering from millibottlenecks instead of available healthy Tomcats. We zoom into a period in which only Tomcat1 has a millibottleneck to see how the total_request policy deals with the millibottleneck. Recall that total_request policy will send all the requests to the Tomcat with the millibottleneck. To verify the instability, we plot one Apache server's workload distribution during the period in which Tomcat 1 has a millibottleneck, as in Figure 6(c). The 4 Apaches have the same workload distribution pattern:

- 5.00s-5.30s (phase 1): In this period, there is no millibottleneck, and the load balancer distributes the workload evenly among the Tomcats.
- 5.35s-5.40s (phase 2): When Tomcat2, 3 and 4 are all idle, all the requests are routed to the Tomcat that has a millibottleneck (Tomcat1).
- 5.45s-5.60s (phase 3): Tomcat1 recovers from the millibottleneck, and almost all of the requests are assigned to Tomcat 2, 3 and 4, to compensate for the uneven workload distribution during the period when all the requests were sent to Tomcat1. We call this period, the recovering period.

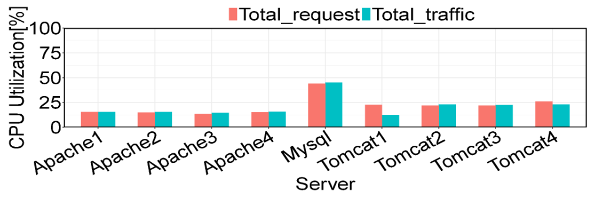
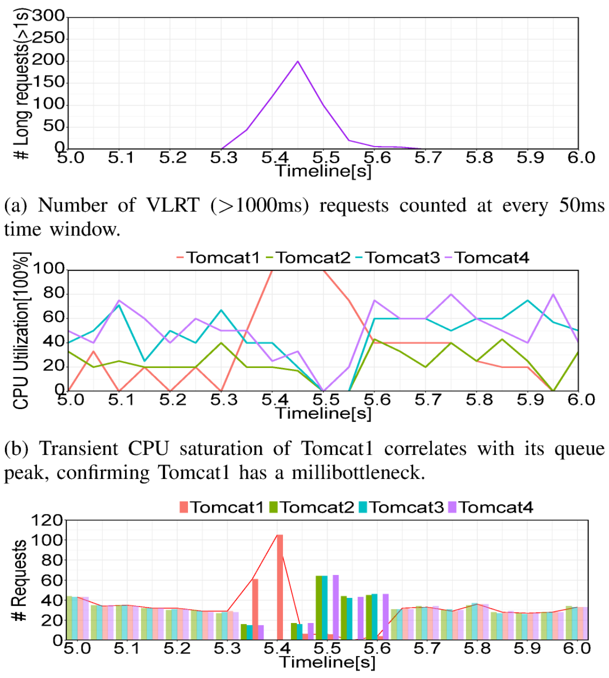


Fig. 5: Average CPU usage among component servers under the total_request and total_traffic policies.



(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.

(b) Transient CPU saturation of Tomcat1 correlates with its queue peak, confirming Tomcat1 has a millibottleneck.

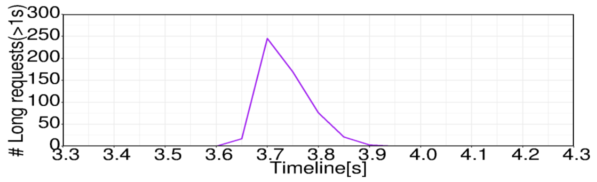
(c) Apache1 workload distribution confirms the load balancing instability: requests were sent to the candidate suffering from the millibottleneck.

Fig. 6: VLRT requests are caused by millibottlenecks, and amplified by the total_request policy instability

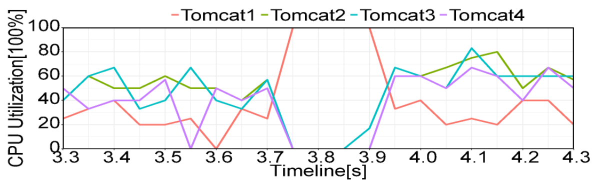
- 5.65s-6.00s (phase 4): No millibottleneck happens, and the load balancer functions as designed.

In the period when Tomcat 1 has a millibottleneck, all requests are sent to it, causing the requests to queue in Tomcat 1. If the requests were assigned to other Tomcats without millibottlenecks, we would expect the queue length peak of Tomcat 1 to be much lower. We would also expect the queue length peaks of Apache tier would be much lower when considering the effect of queue amplification, as mentioned in Section III-B. So the millibottleneck alone didn't create all the VLRT requests, the scheduling issue amplified the magnitude of the VLRT requests.

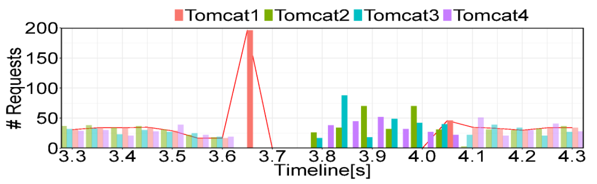
The total_traffic policy has the same instability as the total_request policy. To capture the scheduling issue visually, we plot the workload distribution of 4 Apaches during a period in which one Tomcat has the millibottleneck, as shown in Figure 7(c). By comparing it with the queue length graph in Figure 7(a) and the CPU utilization graph in Figure 7(b), we found all of the requests get routed to the Tomcat that has a



(a) Number of VLRT (>1000ms) requests counted at every 50ms time window.



(b) Transient CPU saturation of Tomcat1 correlates with its queue peak, confirming Tomcat1 has a millibottleneck.



(c) Apache1 workload distribution visually verified the load balancing instability: requests were sent to the candidate with the millibottleneck.

Fig. 7: VLRT requests are caused by millibottlenecks, and amplified by the total_traffic policy instability.

millibottleneck until the millibottleneck is resolved.

IV. Implementation Limitations of Load Balancers and Remedies

Millibottlenecks are very short and difficult to diagnose, they are usually neglected in the load balancer implementation. The preceding implementation assumes Tomcat servers are in stable state, however, millibottlenecks break down the assumptions

A. 3-State Assumption

We begin discussing of the limitations of load balancing by reviewing their implementation (mechanisms). Specifically, the Apache load balancers assume that Tomcat servers are in one of three states during normal execution (not in recovery):

1) Available: the server is able to process the request.
2) Busy: all connections to the server are in use for requests.
3) Error: the server encountered an internal error and is not reachable.

First, the load balancer chooses the available server with the lowest (best) lb_value. Second, the load balancer attempts to get the chosen server to receive the request (called an endpoint). Servers that fail to return an endpoint are moved from the Available state to the Busy state. Third, the load balancer retries (to get an endpoint from) the Busy server. If the retries fail after a specified number, the Busy server is moved to the Error state. The 3-state assumption is very reasonable under the assumption of a stable system state. However, the three states become insufficient when we take millibottlenecks into account.

B. Millibottleneck is mistakenly treated as "Available"

As stated earlier, the millibottlenecks are very short resource bottlenecks on the order of tens to hundreds of milliseconds. The servers are effectively unavailable during periods with the millibottlenecks, but then they become available again after the millibottlenecks resolve.

Algorithm 1: Pseudo code of *get_endpoint*

```
while ((retry * JK_SLEEP_DEF) < aw → cache_acquire_timeout) do
    Iterate through the thread pool, try to find connected endpoint;
    if no connected endpoint found then
        Iterate through the thread pool again to use the first free one;
    if get free endpoint then
        return true;
    else
        retry++;
        sleep(JK_SLEEP_DEF);
return false;
```

The current implementation of Apache load balancer (pseudo code shown in Algorithm 1 keeps checking if the candidate server has a free endpoint until a timeout elapses. If the candidate server doesn't have a free endpoint, it will sleep for a very short period and check again. The default value of cache_acquire_timeout is 300ms and the default sleep time is 100ms. When the load balancer checks for a free endpoint, it does not update the candidate's state or lb_value, while it is waiting for a free endpoint. That is, the server maintains the Available state during this waiting period. This is OK for a permanent failure, since the short waiting period does not impact the ultimate state assigned by the load balancer to the candidate. However, millibottlenecks will lead the load balancer to consider the server to be Available or Busy.

C. Remedy: Millibottleneck as Busy State

To correct this situation, we modified the source code of get_endpoint to take millibottleneck state into consideration. We adopted a conservative approach: treat millibottleneck state as busy state. The idea is very straightforward: when the load balancer tries to find a free endpoint from the candidate, if the candidate cannot respond, the load balancer should skip it and move it to busy state instead of continuing to check it for a very short period.

We used the conservative approach for two reasons: First, it is hard to distinguish millibottleneck from permanent failure [20], and the conservative approach could ensure reliability. Secondly, to handle the latency and throughput demands

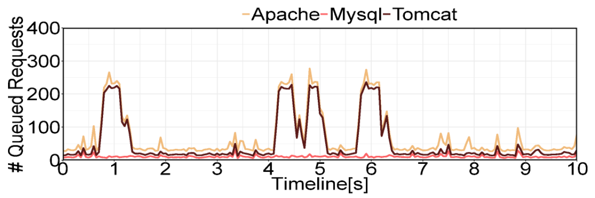


Fig. 8: Queued requests in Apache, Tomcat and MySQL tier under total_request policy with modified get_endpoint. Our remedy at mechanism reduced the queued request by 75%.

of large-scale web services, the load balancer should make decisions quickly.

In the first test, we ran the modified mod_jk under the same workload as previous experiments and configured the load balancing policy to total_request. To verify the effectiveness of the modified get_endpoint, we plot the queue length graph firstly in Figure 8. We see the queue lengths of Tomcat and Apache tier are much lower than original total_request policy. This is because the modified total_request policy avoids sending requests to Tomcat with the millibottleneck, the reason behind it is that the Tomcat with the millibottleneck will not become the candidate, so all request were sent to the available Tomcats.

Workload distribution graph is a more intuitive way to verify the effectiveness of modified get_endpoint. We zoom into a period during which one Tomcat had millibottleneck, as shown in Figure 9(a), a queue spike of Tomcat 2 suggests that it had the millibottleneck. Note that the number of queued requests in Tomcat 2 is 200, which is 1/4 of the original total_request policy. Because of space constraints and the 4 Apaches have the same workload distribution pattern, we only show Apache1's workload load distribution graph in Figure 9(b). We focus on the period in which Tomcat 2 had the millibottleneck[0.80s - 1.15s], we see all the requests were routed to the Tomcats without millibottleneck.

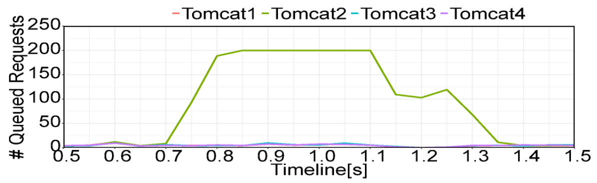
V. Policy Limitations Of Load Balancers and Remedies

A. *Make Decision Based on Cumulative Resource Utilization*

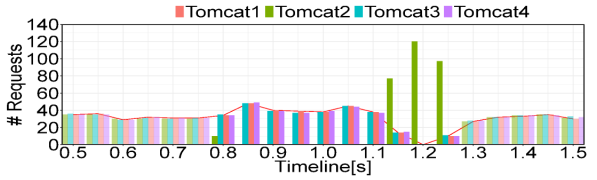
The first policy we study is total_request policy, which says to choose the best Tomcat server based on the fewest total number of requests. Note that lb_value here represents the number of request a candidate has served. Pseudo code for calculating the lb_value under the total_request policy is shown below:

Algorithm 2: Total_request Policy

```
Pick $Tomcat_i$;
if $Tomcat_i$ has a free endpoint to serve the request then
    $Tomcat_i.lb\_value += lb\_mult$;
    Send the request to $Tomcat_i$;
    Receive the response from $Tomcat_i$;
```



(a) Queued requests in Tomcat servers, the small peak in Tomcat2's queue suggests Tomcat2 has a millibottleneck.



(b) Apache1 Workload distribution: after overcoming limitations at mechanism level, the total_request policy doesn't send requests to the candidate with millibottlenecks.

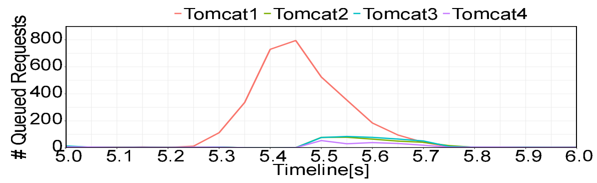
Fig. 9: Remedy at mechanism level can avoid the load balancer instability.

The load balancer picks the candidate with the lowest lb_value, then it queries the candidate by calling the function get_endpoint to determine if it has a free endpoint to serve the request. If the candidate is available, Apache updates its lb_value. Total_traffic policy follows the same procedure as total_request policy, but it uses a different formula to calculate lb_value, which represents number of messages. The pseudo code for the total_traffic policy is shown below:

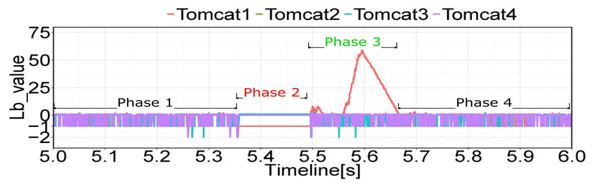
Algorithm 3: Total_traffic Policy

```
Pick $Tomcat_i$;
if $Tomcat_i$ has a free endpoint to service the request then
    Send the request to $Tomcat_i$;
    Receive the response from $Tomcat_i$;
    $Tomcat_i.lb\_value+ =$(read + write sizes of the request)$*lb\_mult$;
```

Due to the mechanism limitation, the lb_value of candidate suffering from millibottlenecks is valid. Due to the policy limitation, its lb_value is the lowest among candidates until the millibottleneck is resolved because other healthy Tomcats' lb_values keep increasing because they can process requests. Firstly, we measured the lb_value under the total request policy to confirm the instability. The total_request policy always picks the candidate with the lowest lb_value, which is the lowest number of completed requests in its semantic. We instrumented code to obtain the lb_value of each candidate. We choose the lb_value of one Tomcat without millibottleneck (Tomcat2) to draw a more clear distinction between the values. Compared to the queue length in Figure 10(a), we observe the common pattern of lb_value changes among the 4 Apaches. Because of space constraints, we only present the lb_value changes in the first Apache in Figure 10(b). The pattern is shown below:

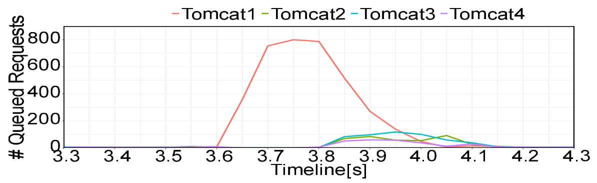


(a) Queued requests in each Tomcat server, the huge peak suggests a millibottleneck happens in Tomcat1.

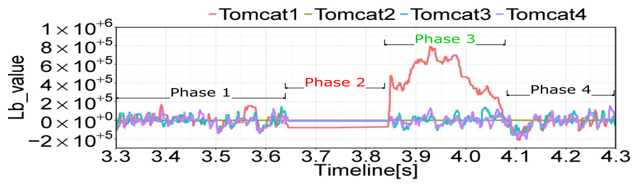


(b) Lb_values of 4 Tomcats: the candidate suffering from a millibottleneck has the lowest lb_value, causing all the requests being sent to it.

Fig. 10: Policy limitations of the total_request policy lead to the load balancer instability.



(a) Queued requests in each Tomcat server, the huge peak suggests a millibottleneck happens in Tomcat1.



(b) Lb_values of 4 Tomcats: the candidate suffering from a millibottleneck has the lowest lb_value, causing all the requests being sent to it.

Fig. 11: Policy limitations of the total_traffic policy lead to the load balancer instability.

- During the normal (without millibottlenecks) periods (phase 1 and phase 4): the lb_value of the 4 Tomcats are identical (i.e., the difference between them is at most 1) since the total_request policy evenly distributes the requests among the 4 Tomcats.
- During the period in which Tomcat1 has the millibottleneck (phase 2), the lb_value of Tomcat1 is the lowest among the 4 Tomcats, which explains why it was picked. As shown in Figure 10(b), the red line is one lower than the other lines in phase 2.
- During the recovery period (phase 3), the lb_value of Tomcat1 is the highest among the 4 Tomcats, as shown by the red peak in Figure 10(b). In phase 2, each Apache accumulates tens of outgoing requests to directed Tomcat1 while no request was sent to other candidates. After Tomcat1 becomes available, it starts to process the requests that have accumulated in phase 2. The other candidates have no such accumulation, which explains why Tomcat1's lb_value increases faster than the other candidates.

The second policy we study is total_traffic policy. We verified the policy limitation of the total_traffic policy by plotting the lb_values of the four candidates. The total_traffic policy's pattern is similar to the total_request policy's: the candidate experiencing a millibottleneck has the lowest lb_value. Recall that the total_traffic policy also picks the candidate with the lowest lb_value. In this setting, lb_value is the sum of the read and write sizes of all of the requests that a candidate has served. We choose the lb_value of one Tomcat without a millibottleneck (Tomcat2) to establish a baseline against with which to compare. We observe the common pattern among the changes in lb_value among the 4 Apaches. Because of space constraints, we only show the first one in Figure 11(b) and don't discuss the details of the pattern here.

The load balancers made the following assumptions: workload is stable and there is no 100% bottleneck on the Tomcat server. These load balancing policies making decisions based on accumulated resource utilization can work well in the absence of millibottlenecks; however, they suffer from limitations in their implementations when millibottlenecks occur. The policy limitations can make the instability even more severe. In the original total_request and traffic policies, the candidate experiencing a millibottleneck is treated as available, and the lb_value of that candidate will not be updated due to the millibottleneck. In contrast, those available candidates' lb_value will keep increasing. This leads the candidate with a millibottleneck to have the lowest lb_value, Accordingly, the load balancer proceeds to send all requests to the candidate with millibottleneck, causing all requests to queue in that candidate.

B. *Make Decision Based on Current State*

These load balancing policies making decisions based on accumulated resource utilization are designed to behave stable under different conditions. The problem faced with these policies is that they are linear, they are not adaptive to changes. To overcome the limitation, the load balancer can acquire additional state information and attempt to make optimistic scheduling decisions based on the current state. Here we introduce a policy using current information to make scheduling decision: current_load policy, the core idea is that it picks the candidate with the least current workload. Pseudo code for current_load policy is shown below, it keeps track of how many requests each candidate is currently assigned at present.

Current_load policy has two advantages: First, it is more adaptive to the millibottlenecks. The candidate suffering from millibottlenecks is unlikely to process the request. Compared to available candidates, it has more requests being served from the perspective of the load balancer. Second, it is robust to the mechanism limitations. Even though Apache could be stuck in calling get_endpoint on the Tomcat with the millibottleneck, the lb_value of the candidate with the millibottleneck remains the highest among the candidates. Current_load policy doesn't

Algorithm 4: Current_load Policy

```
Pick $Tomcat_i$;
if $Tomcat_i$ has a free endpoint to service the request then
    $Tomcat_i.lb\_value+ = lb\_mult$;
    Send the request to $Tomcat_i$;
    Receive the response from $Tomcat_i$;
    if $Tomcat_i.lb\_value \geq lb\_mult$ then
        $Tomcat_i.lb\_value- = lb\_mult$;
    else
        $Tomcat_i.lb\_value = 0$;
```

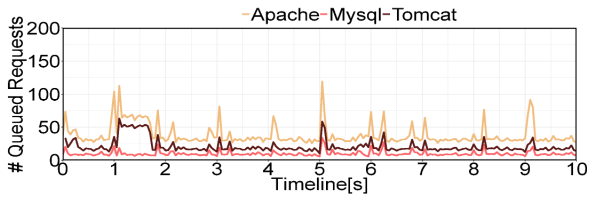


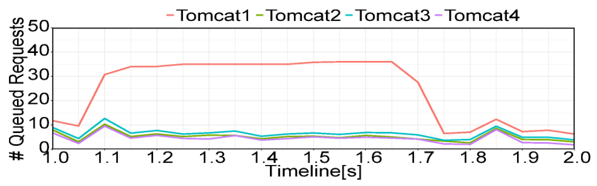
Fig. 12: Queued requests in Apache, Tomcat and MySQL tier under current_load policy. The absence of huge queue peak in the presence of millibottlenecks suggests our remedy at policy level can avoid the scheduling instability.

rely on the 3-stable-state assumption: lb_value of the candidate with the millibottleneck will be the highest among the candidates, which means the candidate with the millibottleneck will not be picked. The implementation limitation of get_endpoint will proceed only after the candidate with millibottleneck is picked. So current_load policy can alleviate the implementation limitations at mechanism level. The shortcoming of current_load policy is rapidly changing system state may cause the policy to react in an unstable manner.

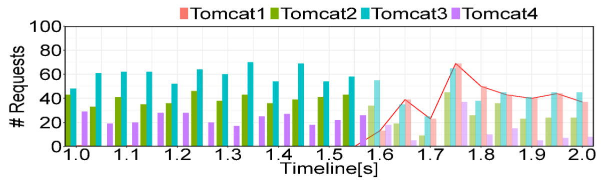
We then use experimental data to show current_load policy is more adaptive to the millibottlenecks. We ran the same workload as total_request and total_traffic policy. First, we examine the queue length graph in Figure 12, we see there is barely any huge spike in Tomcat tier, the reason is that current_load policy is able to avoid sending requests to Tomcat with the millibottleneck. We also observe that the number of spikes of Apache tier is less than total_request and total_traffic policy, this could be explained by the disappearance of queue amplification starting from Tomcat tier. The millibottlenecks are inevitably in Tomcat tier, current_load policy can reduce the magnitude of the VLRT requests by avoiding the scheduling issue.

C. Remedy at Mechanism Level

To further verify the effectiveness of current_load policy, we want to see how current_load policy distributes the workload among the candidates when the millibottleneck occur. We zoom into a period during which one Tomcat has a millibottleneck, as shown in Figure 13(a), a small spike of Tomcat 1 indicates it had millibottleneck. We found a small period of



(a) Queued requests in Tomcat servers, the small queue length peak suggests Tomcat1 has a millibottleneck.



(b) Apache1 Workload distribution: the current_load policy sends all requests to the available candidates instead of the candidates suffering from millibottlenecks.

Fig. 13: Workload distribution further confirms the current_load policy can avoid the scheduling instability.

iowait saturation correlating to the spike, confirming that the millibottleneck is caused by flushing dirty pages. The Tomcat spike is not very obvious, less than 40 requests are queued when the millibottleneck happened, which is not comparable with huge and sharp Tomcat spike under total_request and total_traffic policy. Because of space constraints and the 4 Apaches have the same workload distribution pattern, we only show Apache1's workload load distribution graph in Figure 13. We focus on the period in which Tomcat 1 has the millibottleneck[1.00s - 1.55s], we see all the requests are routed to the Tomcats without millibottlenecks.

VI. Summary of Comparison

VLRT requests are caused by millibottlenecks, amplified by the scheduling issue. We attribute the majority of VLRT requests to three factors: (a) number of the millibottleneck; (b) severity of the millibottleneck and (c)scheduling policy, more concretely, whether the load balancer can avoid sending requests to the Tomcat with the millibottleneck. Since experiment duration, flushing dirty page interval and the total size of dirty pages are almost fixed, we think factor (a) and (b) are fixed across the tests. Such that factor (c) determines the performance. Even millibottlenecks are inevitable, load balancers are supposed to avoid sending requests to the Tomcat with millibottleneck, minimizing the effect of millibottleneck on performance. The combination of limitations at policy and levels will lead to the scheduling instability. Overcoming limitations at least at one level guarantee requests will not be sent to the candidate with millibottlenecks.

To measure the effect of scheduling instability on performance, we first conduct a comparison of the average response time, as shown in Table I. Our remedy at policy level, current_load policy, can improve average response time by 12x and 15x compared to total_request policy and total_traffic policy respectively. Current_load policy can avoid the scheduling instability even though the implementation limitations still

Policy	# Total Requests	Average Response Time (ms)	% VLRT requests (>1000 ms)	% Normal requests (<10 ms)
Original total_request	1791857	41.00	5.33%	88.85%
Original total_traffic	1789493	55.50	6.89%	85.55%
Current_load	1801765	3.62	0.21%	96.70%
Total_request with modified get_endpoint	1800562	4.87	0.55%	95.82%
Total_traffic with modified get_endpoint	1799662	5.87	0.76%	93.93%
Current_workload with modified get_endpoint	1796697	3.60	0.20%	96.67%

TABLE I: Performance of total_request policy, total_traffic policy and our remedies at policy and mechanism levels

exist at mechanism level. Our remedy at mechanism level, modified get_endpoint function, can achieve almost the same performance improvement.

To further verify the effectiveness of our remedies to reduce the VLRT requests amplified by the load balancer instability, we investigate the percentage of VLRT requests (>1000ms), as shown in Table I. We found that current_load policy and fixed get_endpoint reduce the number of the VLRT requests significantly. The percentages of VLRT requests under the total_request policy and total_traffic policy were 5.3% and 6.9%, respectively. Current_load policy, total_request policy with modified get_endpoint and total_traffic policy with modified get_endpoint reduces the percentage of VLRT requests to 0.2%, 0.55% and 0.76% respectively. These VLRT requests are only caused by millibottlenecks. While under total_request and total_traffic policy, the VLRT requests are caused by millibottlenecks and amplified by scheduling issue. We can draw a conclusion based on the comparison: scheduling instability is the dominating factor that leads to the VLRT requests (more that 96% of VLRT requests are caused by scheduling issue). The current_load policy with modified get_endpoint is the case where we overcome limitations at both policy and mechanism levels, but will not gain further improvement because they achieve the same goal.

VII. Related Work

There are efforts focusing on exploring sources of poor tail latency in high large scale distributed applications [12], [6], [7], [29], [21], [22], [27], [26], [28], [24], [30]. Dean et al. [12] present their approaches to bypass/mitigate tail latency in Google's large-scale software application. PriorityMeister [30] automatically and proactively configures workload priorities and rate limits across multiple stages to meet tail latency SLOs for shared networked storage. Suresh et al. [24] propose an adaptive replica selection mechanism to reduce tail latencies, the core idea is using a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control. Bobtail [29] proactively detects and avoids these bad neighboring VMs to avoid long tail problem caused by co-scheduling of CPU-bound and latency-sensitive tasks.

A plethora of approaches are proposed to diagnose performance problems. Li et al. [22] attempt to identify the hardware, operating system, and application-level sources of poor tail latency in high throughput servers executing on multi-core machines. Wang et al. [27] propose a statistical correlation analysis between a server's fine-grained throughput and concurrent jobs in the server to infer the server's real-time performance state. Cohen et al. [11] use a class of probabilistic models (Tree-Augmented Bayesian Networks) to identify combinations of system-level metrics and threshold values that correlate with SLO violations. Chow et al. [10] analyze end-to-end performance of large-scale Internet services through 3 steps: [1] generating a causal model of system behavior via reasoning over software component logs, [2] generating potential hypotheses about program behavior, [3] rejecting hypotheses contradicted by the empirical observations. Aguilera et al. [5] propose approaches to infer the dominant causal paths trough a distributed system from the traces without modifying the system or having semantic knowledge about it. Koskinen et al. [18] obtain precise traces for black-box system without application-specific instrumentation, however, it relies on knowledge of protocols to isolate events or requests. Chopstix [8] collects profiles of low-level OS events at the granularity of executables, procedures and instruction. Then these events are reconstructed to diagnose problems in a large-scale production system.

There is a long history of work in academia and industry to study load balancer [9], [15], [14], [25]. However, these previous works assume the servers are in stable state: either available or failed. Their goal is to achieve good response time and resource utilization. In contrast to these works, our work deals with avoiding long tails in response times in the presence of millibottlenecks. To the best of our knowledge, our paper is the first to systematically analyze load balancer performance in the presence of millibottlenecks. To achieve low latency, Sparrow [23] uses the similar conservative mechanism as we propose when scheduler failure happens: if the scheduler has failed, the client connects to the next scheduler in the list.

VIII. Conclusions

Load balancers in N-tier systems have proven to work well in stable environments. However, we have identified several limitations of some load balancing policies and mechanisms in the presence of millibottlenecks. Our experimental results demonstrate the load balancers instability, where new requests are sent to the candidate suffering from millibottlenecks instead of idle ones, amplifying the magnitude of very long response time request caused by the millibottlenecks.

In this paper, we have found the load balancer instability is caused by limitations at both mechanisms and policy levels. At mechanism level, millibottlenecks are often mistakenly treated as available. To overcome the limitation, we can treat millibottlenecks as busy state. At policy level, policies making decisions based on accumulated resource utilization cannot react to millibottlenecks. Our remedy is using policies that make scheduling decision based on candidate's current state.

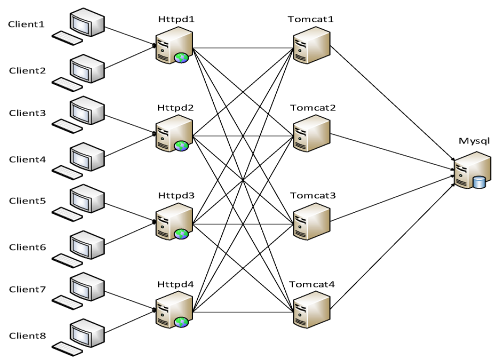


Fig. 14: Topology

We have shown the two remedies can improve the RUBBoS application's response times by a factor of 12 when facing the millibottlenecks caused by flushing dirty pages. The RUBBoS results also demonstrate the effectiveness of our remedies in reducing VLRT requests.

Millibottlenecks are difficult to detect, and it is infeasible to eliminate all of them. Other load balancers in N-tier systems can take advantage of our remedies to shorten the latency tail caused by scheduling instability when facing millibottlenecks caused by other resource shortage. In general, when millibottlenecks happen, we advocate adaptive policies that take into account the current state of servers, with appropriate mechanisms that treat millibottlenecks as unavailable.

IX. Acknowledgments

This research has been partially funded by National Science Foundation by CISE's SAVI/RCN (1402266, 1550379), CN-S (1421561, 1566443), CRISP (1541074), SaTC (1564097) programs, an REU supplement (1545173), Louisiana Board of Regents under grant LEQSF(2015-18)-RD-A-11, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

Appendix

We ran the experiments on Emulab [1], we used 18 d710 nodes to run the RUBBoS benchmark, including 1 control node, 8 client nodes, 4 web servers, 4 application servers and 1 database server, the topology is shown in Figure 14. The first two client nodes send requests to the first web server, the third and fourth client node send requests to the second web server, etc. Each web server communicates with 4 application servers via mod_jk [3]. Table II summarizes the software stack and hardware configuration we used throughout our evaluation. The configurations for each tier can be found in Table III.

Web Server	Apache Httpd 2.2.22
Application Server	Apache Tomcat 5.5.17
Database Server	Mysql 5.5.17
Java	jdk 7
Tomcat Connectors	mod_jk 1.2.32
Operating System	Fedora 15(Kernel 3.3)

CPU	Intel Xeon E5530, 2.40GHz Quad-Core
Memory	12 GB
HDD	WD SATA 7,200 RPM, 500GB
Network Link	1Gbps

TABLE II: Software Stack & Hardware Configuration

Tier	Parameter name	Value
Apache	MaxClients	200
	ThreadsPerChild	100
	WorkerConnectionPoolSize	25
Tomcat	maxThreads	210
DB connections	Total	48
	each Servlet	6
Mysql	Query Cache Size	10MB

TABLE III: Configurations of Major Software Resources

References

[1] Emulab - network emulation testbed. https://www.emulab.net//.
[2] Rubbos: Bulletin board benchmark. http://jmob.ow2.org/rubbos.html/.
[3] Tomcat connectors (mod_jk). https://tomcat.apache.org/download-connectors.cgi/.
[4] Iotop. "http://guichaz.free.fr/iotop/".
[5] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 74–89, 2003.
[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
[7] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 253–266, 2012.
[8] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 103–116, 2008.
[9] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May 1999.
[10] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 217–231, Berkeley, CA, USA, 2014. USENIX Association.
[11] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 231–244, 2004.
[12] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
[13] 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. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.
[14] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract). In *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '85, pages 1–3, New York, NY, USA, 1985. ACM.
[15] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5):662–675, May 1986.

[16] Greg Linden. Make Data Useful. "http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf".

[17] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 959–967, New York, NY, USA, 2007. ACM.

[18] E. Koskinen and J. Jannotti. Borderpatrol: Isolating events for black-box tracing. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 191–203, 2008.

[19] C. A. Lai, Q. Wang, J. Kimball, J. Li, J. Park, and C. Pu. Io performance interference among consolidated n-tier applications: Sharing is better than isolation for disks. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing*, CLOUD '14, pages 24–31, Washington, DC, USA, 2014. IEEE Computer Society.

[20] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.

[21] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, 2014.

[22] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. Technical Report UW-CSE14-04-01, Department of Computer Science & Engineering, University of Washington, April 2014.

[23] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[24] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 513–527, Berkeley, CA, USA, 2015. USENIX Association.

[25] Y. M. Teo and R. Ayani. Comparison of load balancing strategies on cluster-based web servers. *Simulation*, 77(5-6):185–195, 2001.

[26] Q. Wang, Y. Kanemasa, C.-A. Li, Jack Lai, M. Matsubara, and C. Pu. Impact of dvfs on n-tier application performance. In *Proceedings of ACM Conference on Timely Results in Operating Systems (TRIOS 2013)*, pages 33–42, 2013.

[27] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. Detecting transient bottlenecks in n-tier applications through fine-grained analysis. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS 2013)*, pages 31–40, 2013.

[28] Q. Wang, Y. Kanemasa, J. Li, C.-A. Lai, C.-A. 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, Oct. 2014. USENIX Association.

[29] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 329–342, 2013.

[30] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 29:1–29:14, New York, NY, USA, 2014. ACM.