Bobtail: Avoiding Long Tails in the Cloud

Yunjing Xu, Zachary Musgrave, Brian Noble, Michael Bailey
University of Michigan
{yunjing,ztm,bnoble,mibailey}@umich.edu

Abstract

Highly modular data center applications such as Bing, Facebook, and Amazon's retail platform are known to be susceptible to long tails in response times. Services such as Amazon's EC2 have proven attractive platforms for building similar applications. Unfortunately, virtualization used in such platforms exacerbates the long tail problem by factors of two to four. Surprisingly, we find that poor response times in EC2 are a property of nodes rather than the network, and that this property of nodes is both pervasive throughout EC2 and persistent over time. The root cause of this problem is co-scheduling of CPU-bound and latency-sensitive tasks. We leverage these observations in Bobtail, a system that proactively detects and avoids these bad neighboring VMs without significantly penalizing node instantiation. With Bobtail, common communication patterns benefit from reductions of up to 40% in 99.9th percentile response times.

1 Introduction

Modern Web applications such as Bing, Facebook, and Amazon's retail platform are both interactive and dynamic. They rely on large-scale data centers with many nodes processing large data sets at less than human-scale response times. Constructing a single page view on such a site may require contacting hundreds of services [7], and a lag in response time from any one of them can result in significant end-to-end delays [1] and a poor opinion of the overall site [5]. Latency is increasingly viewed as *the* problem to solve [17, 18]. In these data center applications, the long tail of latency is of particular concern, with 99.9th percentile network round-trip times (RTTs) that are orders of magnitude worse than the median [1, 2, 29]. For these systems, one out of a thousand customer requests will suffer an unacceptable delay.

Prior studies have all targeted dedicated data centers. In these, network congestion is the cause of long-tail behavior. However, an increasing number of Internet-scale applications are deployed on commercial clouds such as Amazon's *Elastic Compute Cloud*, or EC2. There are a variety of reasons for doing so, and the recent EC2 outage [21] indicates that many popular online services rely heavily on Amazon's cloud. One distinction between dedicated data centers and services such as EC2 is the use of *virtualization* to provide for multi-tenancy with some degree of isolation. While virtualization does negatively impact latency overall [24, 19], little is known about the long-tail behavior on these platforms.

Our own large-scale measurements of EC2 suggest that median RTTs are comparable to those observed in dedicated centers, but the 99.9th percentile RTTs are up to four times longer. Surprisingly, we also find that nodes of the same configuration (and cost) can have long-tail behaviors that differ from one another by as much as an order of magnitude. This has important implications, as *good* nodes we measured can have long-tail behaviors better than those observed in dedicated data centers [1, 29] due to the difference in network congestion, while *bad* nodes are considerably worse. This classification appears to be a property of the nodes themselves, not data center organization or topology. In particular, bad nodes appear bad to all others, whether they are in the same or different data centers. Furthermore, we find that this property is relatively stable; good nodes are likely to remain good, and likewise for bad nodes within our five-week experimental period. Conventional wisdom dictates that larger (and therefore more expensive) nodes are not susceptible to this problem, but we find that larger nodes are not always better than smaller ones.

Using measurement results and controlled experiments, we find the root cause of the problem to be an interaction between virtualization, processor sharing, and non-complementary workload patterns. In particular, mixing latency-sensitive jobs on the same node with several CPU-intensive jobs leads to longer-than-anticipated scheduling delays, despite efforts of the virtualization layer to avoid them. With this insight, we develop a simple yet effective test that runs locally on a newborn instance and screens between good and bad nodes. We measure common communication patterns [29] on live EC2 instances, and we show improvement in long-tail behavior of between 7% and 40%. While some limits to scale remain, our system effectively removes this first barrier.

2 Observations

Amazon's Elastic Compute Cloud, or EC2, provides dynamic, fine-grained access to computational and storage resources. Virtualization is a key enabler of EC2. We provide background on some of these techniques, and we describe a five-week measurement study of network latency in several different EC2 data centers. Such latency has both significant jitter and a longer tail than that observed in dedicated data centers. Surprisingly, the extra long tail phenomenon is a property of nodes, rather than topology or network traffic; it is pervasive throughout EC2 data centers and it is reasonably persistent.

2.1 Amazon EC2 Background

Amazon EC2 consists of multiple geographically separated *regions* around the world. Each region contains several *availability zones*, or AZs, that are physically isolated and have independent failure probabilities. Thus, one AZ is roughly equivalent to one data center. A version of the Xen hypervisor [3], with various (unknown) customizations, is used in EC2. A VM in EC2 is called an *instance*, and different types (e.g., small, medium, large, and extra large) of instances come with different performance characteristics and price tags. Instances within the same AZ or in different AZs within the same region are connected by a high-speed private network. However, instances within different regions are connected by the public Internet. In this paper, we focus on network tail latency between EC2 instances in the same region.

2.2 Measurement Methodology

Alizadeh *et al.* show that the internal infrastructure of Web applications is based primarily on TCP [1]. But instead of using raw TCP measurement, we use a TCP-based RPC framework called `Thrift`. `Thrift` is popular among Web companies like Facebook [20] and delivers a more realistic measure of network performance at the application level. To measure application-level round-trip-times (RTTs), we time the completion of synchronous RPC calls—`Thrift` adds about $60\mu s$ of overhead when compared to TCP SYN/ACK based raw RTT measurement. In addition, we use established TCP connections for all measurement, so the overhead of the TCP three-way handshake is not included in the RTTs.

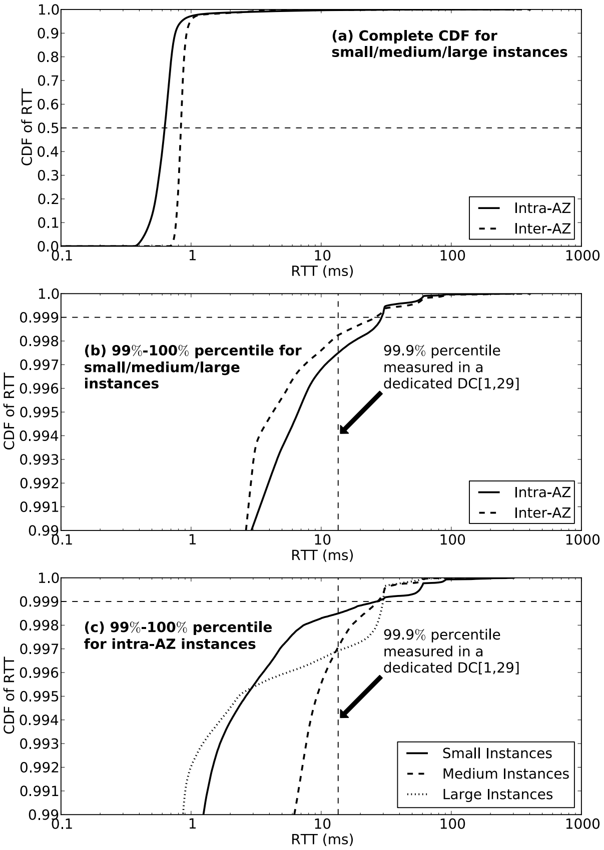

Figure 1: CDF of RTTs for various sized instances, within and across AZs in EC2, compared to measurements taken in a dedicated data center [1, 29]. While the median RTTs are comparable, the 99.9th percentiles in EC2 are twice as bad as in dedicated data centers. This relationship holds for all types of EC2 instances plotted.

2.3 Tail Latency Characterization

We focus on the tail of round-trip latency due to its disproportionate impact on user experience. Other studies have measured network performance in EC2, but they often use metrics like mean and variance to show jitter in network and application performance [24, 19]. While these measurements are useful for high-throughput applications like MapReduce [6], worst-case performance matters much more to applications like the Web that require excellent user experience [29]. Because of this, researchers use the RTTs at the 99th and 99.9th percentiles to measure flow tail completion times in dedicated data centers [1, 29].

We tested network latency in EC2's US east region for five weeks. Figure 1 shows CDFs for both a combination of small, medium, and large instances and for discrete sets of those instances. While (a) and (b) show aggregate measurements both within and across availability zones (AZs), (c) shows discrete measurements for three instance types within a specific AZ.

In Figure 1(a), we instantiated 20 instances of each type for each plot, either within a single AZ or across two AZs in the same region. We observe that median RTTs within a single AZ, at ∼0.6*ms*, compare well to those found within a dedicated data center at ∼0.4*ms* [1, 29], even though our measurement method adds 0.06*ms* of overhead. Inter-AZ measurements show a median RTT of under 1*ms*. However, distances between pairs of AZs may vary; measurements taken from another pair of AZs show a median RTT of around 2*ms*.

Figure 1(b) shows the 99th to 100th percentile range of (a) across all observations. Unfortunately, its results paint a different picture of latency measurements in Amazon's data centers. The 99.9th percentile of RTT measurements is *twice as bad* as the same metric in a dedicated data center [1, 29]. Individual nodes can have 99.9th percentile RTTs up to four times higher than those seen in such centers. Note that this observation holds for both curves; no matter whether the measurements are taken in the same data center or in different ones, the 99.9th percentiles are almost the same.

Medium, large, and extra large instances ostensibly offer better performance than their small counterparts. As one might expect, our measurements show that extra large instances do not exhibit the extra long tail problem (< 0.9*ms* for the 99.9th percentile); but surprisingly, as shown in Figure 1(c), medium and large instances are susceptible to the problem. In other words, the extra long tail is not caused by a specific type of instance: all instance types shown in (c) are equally susceptible to the extra long tail at the 99.9th percentile. Note that all three lines in the figure intersect at the 99.9th line with a value of around 30ms. The explanation of this phenomenon becomes evident in the discussion of the root cause of the long tail problem in § 3.2.

To explore other factors that might create extra long tails, we launch 16 instances within the same AZ and measure the *pairwise* RTTs between each instance. Figure 2 shows measurement results at the 99.9th percentile in milliseconds. Rows represent source IP addresses, while columns represent destination IP addresses.

Were host location on the network affecting long tail performance, we would see a symmetric pattern emerge on the heat map, since network RTT is a symmetric measurement. Surprisingly, the heat map is asymmetric—there are vertical bands which do not correspond to reciprocal pairings. To a large degree, the destination host controls whether a long tail exists. In other words, *the extra long tail problem in cloud environments is a property of nodes, rather than the network.*

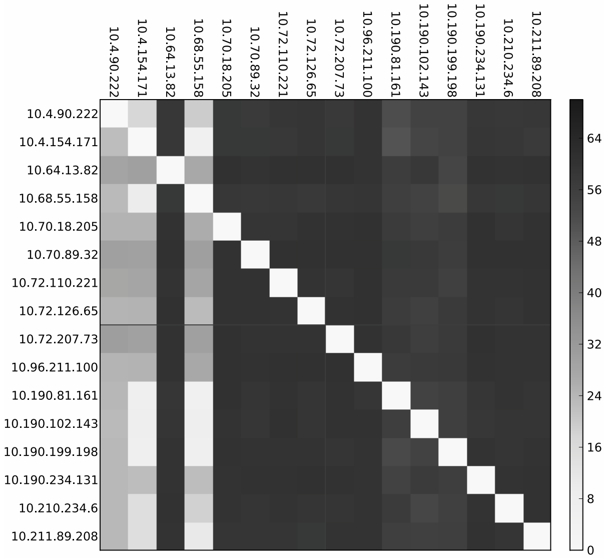

Figure 2: Heat map of the 99.9th percentile of RTTs, shown for 16 small pairwise instances in milliseconds. Bad instances, represented by dark vertical bands, are bad consistently. This suggests that the long tail problem is a property of specific nodes instead of the network.

Interestingly, the data shown in Figure 2 is not entirely bleak: there are *both* dark and light bands, so tail performance between nodes varies drastically. Commonly, RPC servers are allowed only 10*ms* to return their results [1]. Therefore, we refer to nodes that fulfill this service as *good* nodes, which appear in Figure 2 as light bands; otherwise, they are referred to as *bad* nodes. Under this definition, we find that RTTs at the 99.9th percentile can vary by *up to an order of magnitude* between good nodes and bad nodes. In particular, the bad nodes we measured can be two times worse than those seen in a dedicated DC [1, 29] for the 99.9th percentile. This is because the latter case's latency tail is caused by network congestion, whose worst case impact is bounded by the egress queue size of the bottleneck switch port, but the latency tail problem we study here is a property of nodes, and its worst case impact can be much larger than that caused by network queueing delay. This observation will become more clear when we discuss the root cause of the problem in § 3.2.

To determine whether bad nodes are a pervasive problem in EC2, we spun up 300 small instances in each of four AZs in the US east region. We measured all the nodes' RTTs (the details of the measurement benchmarks can be found in § 5.1) and we found 40% to 70% bad nodes within three of the four AZs.

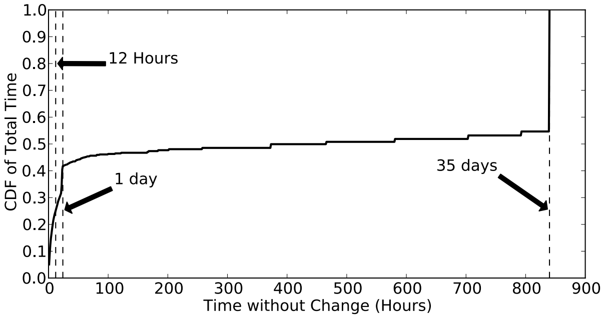

Figure 3: CDF for the time periods during which instances do not switch status between good and bad. This shows that properties of instances generally persist.

Interestingly, the remaining AZ sometimes does not return bad nodes; nevertheless, when it does, it returns 40% to 50% bad nodes. We notice that this AZ spans a smaller address space of only three /16 subnets compared to the others, which can span tens of /16 subnets. Also, its available CPU models are, on average, newer than those found in any of the other AZs; Ou *et al.* present similar findings [16], so we speculate that this data center is newly built and loaded more lightly than the others. We will discuss this issue further in conjunction with the root cause analysis in § 3.

We also want to explore whether the long latency tail we observe is a persistent problem, because it is a property defined by node conditions rather than transient network conditions. We conducted a five week experiment comprised of two sets of 32 small instances: one set was launched in equal parts from two AZs, and one set was launched from all four AZs. Within each set, we selected random pairs of instances and measured their RTTs throughout the five weeks. We observed how long instances' properties remain static—either good or bad without change—to show the persistence of our measurement results.

Figure 3 shows a CDF of these stable time periods; persistence follows if a large percentage of total instance time in the experiment is comprised of large time periods. We can observe that almost 50% of the total instance time no change has been witnessed, 60% of time involves at most one change per day, and 75% of time involves at most one change per 12 hours. This result shows that the properties of long tail network latency are generally persistent.

The above observation should be noted by the following: *every night*, *every instance* we observe in EC2 experiences an abnormally long latency tail for several minutes at midnight Pacific Time. For usually bad instances this does not matter; however, usually good instances are forced to change status at least once a day. Therefore, the figures we state above can be regarded as overestimating the frequency of changes. It also implies that the 50% instance time during which no change has been witnessed belongs to bad instances.

3 Root Cause Analysis

We know that the latency tail in EC2 is two to four times worse than that in a dedicated data center, and that as a property of nodes instead of the network it persists. Then, what is its root cause? Wang *et al.* reported that network latency in EC2 is highly variable, and they speculated that virtualization and processor sharing make up the root cause [24].

However, the coexistence of good and bad instances suggests that processor sharing under virtualization is *not sufficient* to cause the long tail problem by itself. We will show in this section that only *a certain mix of workloads* on shared processors can cause this problem, and we demonstrate the patterns of such a bad mix.

3.1 Xen Hypervisor Background

To fully understand the impact of processor sharing and virtual machine co-location on latency-sensitive workloads under Xen, we must first present some background on the hypervisor and its virtual machine scheduler. The Xen hypervisor [3] is an open source virtual machine monitor, and it is used to support the infrastructure of EC2 [24]. Xen consists of one privileged virtual machine (VM) called dom0 and multiple guest VMs called domUs. Its VM scheduler is credit-based [28], and by default it allocates 30*ms* of CPU time to each virtual CPU (VCPU); this allocation is decremented in 10*ms* intervals. Once a VCPU has exhausted its credit, it is not allowed to use CPU time unless no other VCPU has credit left; any VCPU with credit remaining has a higher priority than any without. In addition, as described by Dunlap [8], a lightly-loaded VCPU with excess credit may enter the BOOST state, which allows a VM to automatically receive first execution priority when it wakes due to an I/O interrupt event. VMs in the same BOOST state run in FIFO order. Even with this optimization, Xen's credit scheduler is known to be unfair to latency-sensitive workloads [24, 8]. As long as multiple VMs are sharing physical CPUs, one VM may need to wait tens of milliseconds to acquire a physical CPU if others are using it actively. Thus, when a VM handles RPC requests, certain responses will have to wait tens of milliseconds before being returned. This implies that *any* VM can exhibit a high maximum RTT.

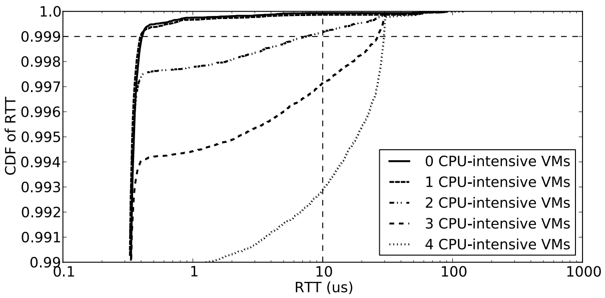

Figure 4: CDF of RTTs for a VM within controlled experiments, with an increasing number of co-located VMs running CPU-intensive workloads. Sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs, but once this condition no longer holds, the long tail emerges.

3.2 Root Cause Explained

If processor sharing under virtualization does not always cause the extra long tail problem, when does it? To answer this question, we conduct five controlled experiments with Xen.

On a four-core workstation running Xen 4.1, dom0 is pinned to two cores while guest VMs use the rest. In all experiments, five identically configured domUs share the remaining two physical cores; they possess equal weights of up to 40% CPU utilization each. Therefore, though domUs may be scheduled on either physical core, none of them can use more than 40% of a single core even if there are spare cycles. To the best of our knowledge, this configuration is the closest possible to what EC2 *small* instances use. Note that starting from Xen 4.2, a configurable rate limit mechanism is introduced to the credit scheduler [28]. In its default setting, a running VM cannot be preempted if it has run for less than 1ms. To obtain a result comparable to the one in this section using Xen 4.2 or newer, the rate limit needs to be set to its minimum of 0.1ms.

For this set of experiments, we vary the workload types running on five VMs sharing the local workstation. In the first experiment, we run the Thrift RPC server in all five guest VMs; we use another non-virtualized workstation in the same local network to make RPC calls to all five servers, once every two milliseconds, for 15 minutes. During the experiment, the local network is never congested. In the next four experiments, we replace the RPC servers on the guest VMs with a CPU-intensive workload, one at a time, until four guest VMs are CPU-intensive and the last one, called the *victim VM*, remains latency-sensitive.

Figure 4 shows the CDF of our five experiments' RTT distributions from the 99th to the 100th percentile for the victim VM. While four other VMs also run latency-sensitive jobs (zero VMs run CPU-intensive jobs), the latency tail up to the 99.9th percentile remains under 1*ms*. If one VM runs a CPU-intensive workload, this result does not change. Notably, even when the victim VM *does share* processors with one CPU-intensive VM and three latency-sensitive VMs, the extra long tail problem is *nonexistent*.

However, the 99.9th percentile becomes *five times* larger once two VMs run CPU-intensive jobs. This still qualifies as a good node under our definition ($< 10ms$), but the introduction of even slight network congestion could change that. To make matters worse, RTT distributions increase further as more VMs become CPU-intensive. Eventually, the latency-sensitive victim VM behaves just like the bad nodes we observe in EC2.

The results of the controlled experiments assert that virtualization and processor sharing are not sufficient to cause high latency effects across the entire tail of the RTT distribution; therefore, much of the blame rests upon co-located workloads. We show that having one CPU-intensive VM is acceptable; why does adding one more suddenly make things five times worse?

There are two physical cores available to guest VMs; if we have one CPU-intensive VM, the latency-sensitive VMs can be scheduled as soon as they need to be, while the single CPU-intensive VM occupies the other core. Once we reach two CPU-intensive VMs, it becomes possible that they occupy both physical cores concurrently while the victim VM has an RPC request pending. Unfortunately, the BOOST mechanism does not appear to let the victim VM preempt the CPU-intensive VMs often enough. Resulting from these unfortunate scenarios is an extra long latency distribution. In other words, *sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs; once this condition no longer holds, the long tail emerges.*

This set of controlled experiments demonstrates that a certain mix of latency-sensitive and CPU-intensive workloads on shared processors can cause the long tail problem, but a question remains: will all CPU-intensive workloads have the same impact? In fact, we notice that if the co-located CPU-intensive VMs in the controlled experiments always use 100% CPU time, the latency-sensitive VM *does not* suffer from the long tail problem—its RTT distribution is similar to the one without co-located CPU-intensive VMs; the workload we use in the preceding experiments actually uses about 85% CPU time. This phenomenon can be explained by the design of the BOOST mechanism. Recall that a VM waking up due to an interrupt may enter the BOOST state if it has credits remaining. Thus, if a VM doing mostly CPU-bound operations decides to accumulate scheduling credits, e.g., by using the sleep function call, it will also get BOOSTed after the sleep timer expires. Then, it may mo-

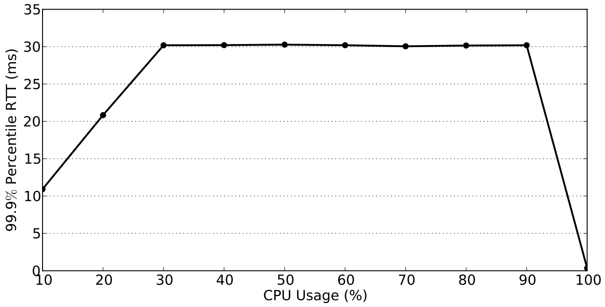

Figure 5: The relationship between the 99.9th percentile RTT for the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM.

nopolize the CPU until its credits are exhausted without being preempted by other BOOSTed VMs, some of which may be truly latency-sensitive.

In other words, the BOOST mechanism is only effective against the workloads that use almost 100% CPU time because such workloads exhaust their credits easily and BOOSTed VMs can then preempt them whenever they want. To study the impact of lower CPU usage, we conduct another controlled experiment by varying the CPU usage of the CPU-intensive workload from 10% to 100% and measuring the 99.9th percentile RTT of the latency-sensitive workload. We control the CPU usage using a command-line utility called cpulimit on a process that otherwise uses 100% CPU time; cpulimit pauses the target process periodically to adjust its average CPU usage. In addition, based on what we learned from the first set of controlled experiments, we only need to use one latency-sensitive VM to share a single CPU core with one CPU-intensive VM and allocate 50% CPU time to each one respectively.

Figure 5 shows the relationship between the 99.9th percentile RTT of the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM. Surprisingly, the latency tail is over 10ms even with 10% CPU usage, and starting from 30%, the tail latency is almost constant until 100%. This is because by default cpulimit uses a 10*ms* granularity: given X% CPU usage, it makes the CPU-intensive workload work Xms and pause $(100-X)ms$ in each 100*ms* window. Thus, when $X < 30$, the workload yields the CPU every Xms, so the 99.9th percentiles for the 10% and 20% cases are close to 10*ms* and 20*ms*, respectively; for $X \geq 30$, the workload keeps working for at least 30*ms*. Recall that the default time slice of the credit scheduler is 30*ms*, so the CPU-intensive workload cannot keep running for more than 30*ms* and we see the flat line in Figure 5. It also explains why the three curves in Figure 1(c) intersect at the 99.9th percentile line. The takeaway is that even if a workload uses as little as 10% CPU time on average, it still can cause a long latency tail to neighboring VMs by using large bursts of CPU cycles (e.g., 10*ms*). In other words, average CPU usage does not capture the intensity of a CPU-bound workload; *it is the length of the bursts of CPU-bound operations that matters*.

Now that we understand the root cause, we will examine an issue stated earlier: one availability zone in the US east region of EC2 has a higher probability of returning good instances than the other AZs. If we break down VMs returned from this AZ by CPU model, we find a higher likelihood of newer CPUs. These newer CPUs should be more efficient at context switching, which naturally shortens the latency tail, but what likely matters more is newer CPUs' possessing six cores instead of four, as in older CPUs that are more common in the other three data centers. One potential explanation for this is that the EC2 instance scheduler may not consider CPU model differences when scheduling instances sensitive to delays. Then, a physical machine with four cores is much more likely to be saturated with CPU-intensive workloads than a six-core machine. Hence, a data center with older CPUs is more susceptible to the problem. Despite this, our root cause analysis always applies, because we have observed that both good and bad instances occur regardless of CPU model; differences between them only change the likelihood that a particular machine will suffer from the long tail problem.

4 Avoiding the Long Tails

While sharing is inevitable in multi-tenant cloud computing, we set out to design a system, Bobtail, to find instances where processor sharing does not cause extra long tail distributions for network RTTs. Cloud customers can use Bobtail as a utility library to decide on which instances to run their latency-sensitive workloads.

4.1 Potential Benefits

To understand both how much improvement is possible and how hard it would be to obtain, we measured the impact of bad nodes for common communication patterns: sequential and partition-aggregation [29]. In the sequential model, an RPC client calls some number of servers in series to complete a single, timed observation. In the partition-aggregation model, an RPC client calls all workers in parallel for each timed observation.

For the sequential model, we simulate workflow completion time by sampling from the measured RTT distributions of good and bad nodes. Specifically, every time, we randomly choose one node out of N RPC servers to request 10 flows serially, and we repeat this $2,000,000$ times. Figure 6 shows the 99th and 99.9th percentile values of the workflow completion time, with an increasing number of bad nodes among a total of 100 instances.

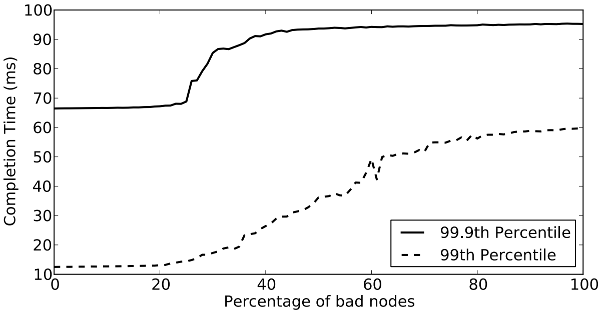

Figure 6: Impact of bad nodes on the flow tail completion times of the sequential model. Bobtail can expect to reduce tail flow completion time even when as many as 20% of nodes are bad.

Interestingly, there is no difference in the tails of overall completion times when as many as 20% of nodes are bad. But the difference in flow tail completion time between 20% bad nodes and 50% bad nodes is severe: flow completion time increases by *a factor of three* at the 99th percentile, and a similar pattern exists at the 99.9th percentile with a smaller difference. This means Bobtail is allowed to make mistakes—even if up to 20% of the instances picked by Bobtail are actually bad VMs, it still helps reduce flow completion time when compared to using random instances from EC2. Our measurements suggest that receiving 50% bad nodes from EC2 is not uncommon.

Figure 7 shows the completion time of the partition-aggregation model when there are 10, 20, and 40 nodes in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes. However, there is less room for error here than in the sequential model: as the system scales up, other barriers present themselves, and avoiding nodes we classify as bad provides diminishing returns. Understanding this open question is an important challenge for us going forward.

4.2 System Design and Implementation

Bobtail needs to be a scalable system that makes accurate decisions in a timely fashion. While the node property remains stable in our five-week measurement, empirical evidence shows that the longer Bobtail runs, the more accurate its result can be. However, because launching an instance takes no more than a minute in EC2, we limit Bobtail to making a decision in under two minutes. Therefore, we need to strike a balance between accuracy and scalability.

A naive approach might be to simply conduct network measurements with every candidate. But however accurate it might be, such a design would not scale well to handle a large number of candidate instances in parallel: to do so in a short period of time would require sending a large amount of network traffic as quickly as possible to all candidates, and the synchronous nature of the measurement could cause severe network congestion or even TCP incast [23].

On the other hand, the most scalable approach involves conducting testing locally at the candidate instances, which does not rely on any resources outside the instance itself. Therefore, all operations can be done quickly and in parallel. This approach trades accuracy for scalability. Fortunately, Figures 6 and 7 show that Bobtail is allowed to make mistakes.

Based on our root cause analysis, such a method exists because the part of the long tail problem we focus on is *a property of nodes instead of the network*. Accordingly, if we know the workload patterns of the VMs co-located with the victim VM, we should be able to predict if the victim VM will have a bad latency distribution locally without any network measurement.

In order to achieve this, we must infer how often long scheduling delays happen to the victim VM. Because the long scheduling delays caused by the co-located CPU-intensive VMs are not unique to network packet processing and any interrupt-based events will suffer from the same problem, we can measure the frequency of large delays by measuring the time for the target VM to wake up from the `sleep` function call—the delay to process the timer interrupt is a proxy for delays in processing all hardware interrupts.

To verify this hypothesis, we repeat the five controlled experiments presented in the root cause analysis. But instead of running an RPC server in the victim VM and measuring the RTTs with another client, the victim VM runs a program that loops to sleep 1*ms* and measures the *wall time* for the `sleep` operation. Normally, the VM should be able to wake up after a little over 1*ms*, but co-located CPU-intensive VMs may prevent it from doing so, which results in large delays.

Figure 8 shows the number of times when the sleep time rises above 10*ms* in the five scenarios of the controlled experiments. As expected, when two or more VMs are CPU-intensive, the number of large delays experienced by the victim VM is *one to two orders of magnitude* above that experienced when zero or one VMs are CPU-intensive. Although the fraction of such large delays is small in all scenarios, the large difference in the raw counts forms a clear criterion for distinguishing bad nodes from good nodes. In addition, although it is not shown in the figure, we find that large delays with zero or one CPU-intensive VMs mostly appear for lengths of around 60*ms* or 90*ms*; these are caused by the 40% CPU cap on each latency-sensitive VM (i.e., when they are not allowed to use the CPU despite its availability). Delays

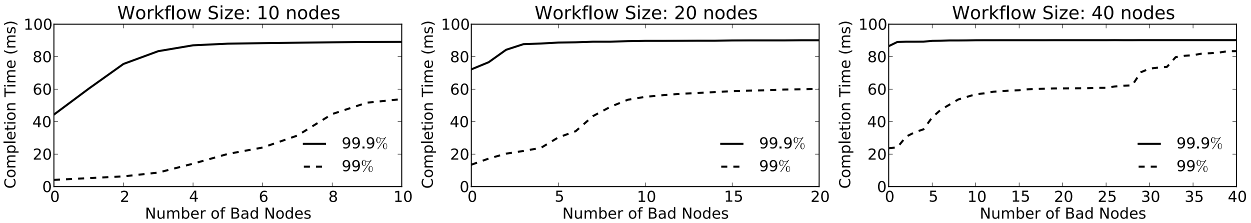

Figure 7: Impact of bad nodes on the tail completion time of the partition-aggregation model with 10, 20, and 40 nodes involved in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes.

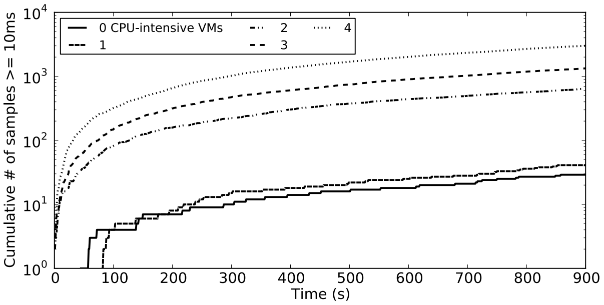

Figure 8: The number of large scheduling delays experienced by the victim VM in controlled experiments with an increasing number of VMs running CPU-intensive workloads. Such large delay counts form a clear criterion for distinguishing bad nodes from good nodes.

Algorithm 1 Instance Selection Algorithm

```
num_delay = 0
for $i = 1 \to M$ do
    sleep for S micro seconds
    if sleep time $\geq$ 10ms then
        num_delay++
    end if
end for
if num_delay $\leq$ LOW_MARK then
    return GOOD
end if
if num_delay $\leq$ HIGH_MARK then
    return MAY USE NETWORK TEST
end if
return BAD
```

experienced in other scenarios are more likely to be below 30*ms*, which is a result of latency-sensitive VMs preempting CPU-intensive VMs. This observation can serve as another clue for distinguishing the two cases.

Based on the results of our controlled experiments, we can design an instance selection algorithm to predict *locally* if a target VM will experience a large number of long scheduling delays. Algorithm 1 shows the pseudocode of our design. While the algorithm itself is straightforward, the challenge is to find the right threshold in EC2 to distinguish the two cases (`LOW_MARK` and `HIGH_MARK`) and to draw an accurate conclusion as quickly as possible (loop size M).

Our current policy is to reduce false positives, because in the partition-aggregation pattern, reducing bad nodes is critical to scalability. The cost of such conservatism is that we may label good nodes as bad incorrectly, and as a result we must instantiate even more nodes to reach a desired number. To return N good nodes as requested by users, our system needs to launch $K * N$ instances, and then it needs to find the best N instances of that set with the lowest probability of producing long latency tails.

After Bobtail fulfills a user's request for N instances whose delays fall below `LOW_MARK`, we can apply the network-based latency testing to the leftover instances whose delays fall between `LOW_MARK` and `HIGH_MARK`; this costs the user nothing but provides further value using the instances that users already paid for by the hour. Many of these nodes are likely false negatives which, upon further inspection, can be approved and returned to the user. In this scenario, scalability is no longer a problem because we no longer need to make a decision within minutes. Aggregate network throughput for testing can be thus much reduced. With this optimization, we may achieve a much lower effective false negative rate, which will be discussed in the next subsection.

A remaining question is what happens if users run latency-sensitive workloads on the good instances Bobtail picked, but those VMs become bad after some length of time. In practice, because users are running network workloads on these VMs, they can tell if any good VM turns bad by inspecting their application logs without any extra monitoring effort. If it happens, users may use Bobtail to pick more good VMs to take the place of the bad ones. Fortunately, as indicated in Figure 3, our five-week measurement shows that such properties generally persist, so workload migration does not need to happen very frequently. In addition, Figures 6 and 7 also indicate that

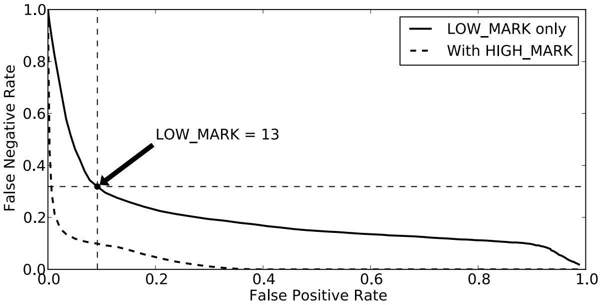

Figure 9: Trade-off between false positive and false negative rates of the instance selection algorithm. Our system can achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3. With the help of network-based testing, the effective false negative rate can be reduced to below 0.1.

even if 20% of instances running latency-sensitive workloads are bad VMs, their impact on the latency distribution of sequential or partition-aggregation workloads is limited.

4.3 Parameterization

To implement Bobtail's algorithm, we need to define both its runtime (loop size M) and the thresholds for the LOW_MARK and HIGH_MARK parameters. Our design intends to limit testing time to under two minutes, so in our current implementation we set the loop size M to be $600K$ sleep operations, which translates to about 100 seconds on small instances in EC2—the worse the instance is, the longer it takes.

The remaining challenge we face is finding the right thresholds for our parameters (LOW_MARK and HIGH_MARK). To answer this inquiry, we launch 200 small instances from multiple availability zones (AZs) in EC2's US east region, and we run the selection algorithm for an hour on all the candidates. Meanwhile, we use the results of network-based measurements as the *ground truth* of whether the candidates are good or bad. Specifically, we consider the instances with 99.9th percentiles under $10ms$ for *all* micro benchmarks, which are discussed in § 5.1, as good nodes; all other nodes are considered bad.

Figure 9 shows the trade-off between the false positive and false negative rates by increasing LOW_MARK from 0 to 100. The turning point of the solid line appears when we set LOW_MARK around 13, which lets Bobtail achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3—a good balance between false positive and false negative rates. Once HIGH_MARK is introduced (as five times LOW_MARK), the effective false negative rate can be reduced to below 0.1, albeit with the help of network-based testing. We leave it as future work to study when we need to re-calibrate these parameters.

The above result reflects our principle of favoring a low false positive. Therefore, we need to use a relatively large K value in order to get N good nodes from $K*N$ candidates. Recall that our measured good node ratio for random instances directly returned by EC2 ranges from 0.4 to 0.7. Thus, as an estimation, with a 0.3 false negative rate and a 0.4 to 0.7 good node ratio for random instances from multiple data centers, we need $K*N*(1-0.3)*0.4 = N$ or $K \approx 3.6$ to retrieve the number of desired good nodes from one batch of candidates. However, due to the pervasiveness of bad instances in EC2, even if Bobtail makes no mistakes we still need a minimum of $K*N*0.4 = N$ or $K = 2.5$. If startup latency is the critical resource, rather than the fees paid to start new instances, one can increase this factor to improve response time.

5 Evaluation

In this section, we evaluate our system over two availability zones (AZs) in EC2's US east region. These two AZs always return some bad nodes. We compare the latency tails of instances both selected by our system and launched directly via the standard mechanism. We conduct this comparison using both micro benchmarks and models of sequential and partition-aggregation workloads.

In each trial, we compare 40 small instances launched directly by EC2 from one AZ to 40 small instances selected by our system from the same AZ. The comparison is done with a series of benchmarks; these small instances will run RPC servers for all benchmarks. To launch 40 good instances, we use $K = 4$ with 160 candidate instances. In addition, we launch four extra large instances for every 40 small instances to run RPC clients. We do this because, as discussed earlier, extra large instances do not experience the extra long tail problem; we therefore can blame the server instances for bad latency distributions.

5.1 Micro Benchmarks

Our traffic models for both micro benchmarks and sequential and partition-aggregation workloads have inter-arrival times of RPC calls forming a Poisson process. For micro benchmarks, we assign 10 small instance servers to each extra large client. The RPC call rates are set at 100, 200, and 500 calls/second. In each RPC call, the client sends an 8-byte request to the server, and the server responds with 2KB of random data. Meanwhile, both requests and responses are packaged with another 29-byte overhead. The 2KB message size was chosen because measurements taken in a dedicated data center in-

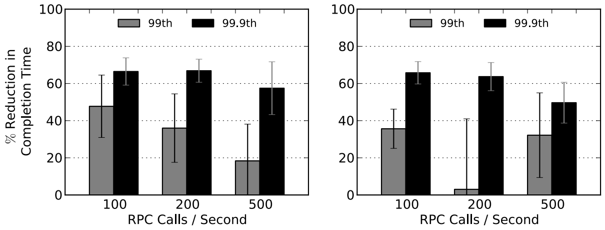

Figure 10: Reduction in flow tail completion time in micro benchmarks by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

dicate that most latency-sensitive flows are around 2KB in size [1]. Note that we do not generate artificial background traffic, because real background traffic already exists throughout EC2 where we evaluate Bobtail.

Figure 10 presents the reductions in completion times for three RPC request rates in micro benchmarks across two AZs. Bobtail reduces latency at the 99.9th percentile from 50% to 65%. In micro benchmark and subsequent evaluations, the mean of reduction percentages in flow completion is presented with a 90% confidence interval.

However, improvements at the 99th percentile are smaller with a higher variance. This is because, as shown in Figure 1, the 99th percentile RTTs within EC2 are not very bad to begin with ($\sim 2.5ms$); therefore, Bobtail's improvement space is much smaller at the 99th percentile than at the 99.9th percentile. For the same reason, network congestion may have a large impact on the 99th percentile while having little impact on the 99.9th percentile in EC2. The outlier of 200 calls/second in the second AZ of Figure 10 is caused by one trial in the experiment with 10 good small instances that exhibited abnormally large values at the 99th percentile.

5.2 Sequential Model

For sequential workloads, we apply the workload model to 20-node and 40-node client groups, in addition to the 10-node version shown in the micro benchmarks. In this case, the client sends the same request as before, but the servers reply with a message size randomly chosen from among 1KB, 2KB, and 4KB. For each workflow, instead of sending requests to all the servers, the client will randomly choose one server from the groups of sizes 10, 20, and 40. Then, it will send 10 synchronous RPC calls to the chosen server; the total time to complete all 10 RPC requests is then used as the workflow RTT. Because of this, the workflow rates for the sequential model are reduced to one tenth of the RPC request rates for micro benchmarks and become 10, 20, and 50 workflows per second.

Figure 11 shows our improvement under the sequential model with different numbers of RPC servers involved. Bobtail brings a 35% to 40% improvement to sequential workloads at the 99th percentile across all experiments, and it roughly translates to an 8ms reduction. The lengths of the confidence intervals grow as the number of server nodes increases; this is caused by a relatively smaller sample space. The similarity in the reduction of flow completion time with different numbers of server nodes shows that the tail performance of the sequential workflow model only depends on the ratio of bad nodes among all involved server nodes. Essentially, the sequential model demonstrates the average tail performance across all server nodes by randomly choosing one server node each time with equal probability at the client side.

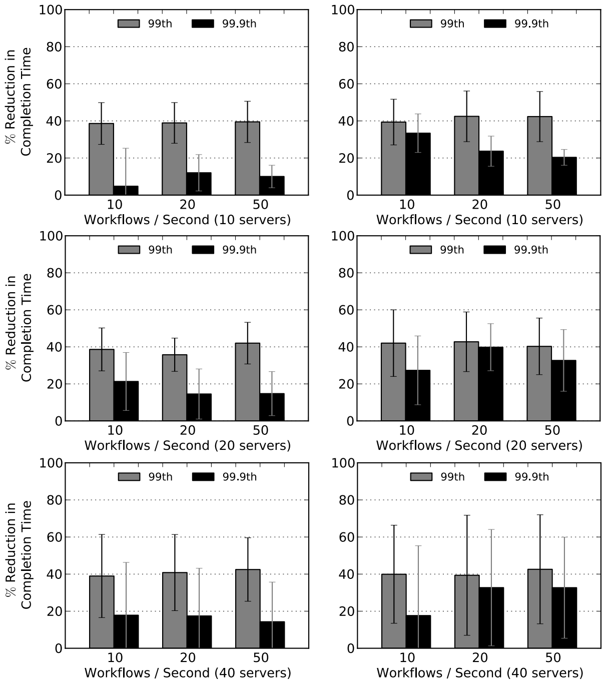

Figure 11: Reduction in flow tail completion time for sequential workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

Interestingly, and unlike in the micro benchmarks, improvement at the 99.9th percentile now becomes smaller and more variable. However, this phenomenon does match our simulation result shown in Figure 6 when discussing the potential benefits of using Bobtail.

5.3 Partition-Aggregation Model

For the partition-aggregation model, we use the same 10, 20, and 40-node groups to evaluate Bobtail. In this case, the client always sends requests to all servers in the group concurrently, and the workflow finishes once the slowest

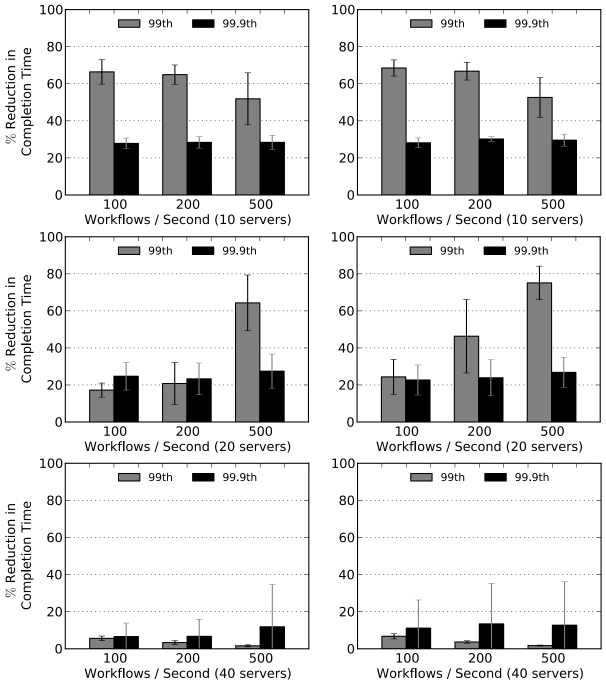

Figure 12: Reduction in flow tail completion time for partition-aggregation workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

RPC response returns; servers always reply with 2KB of random data. In other words, the RTT of the slowest RPC call is effectively the RTT of the workflow. Meanwhile, we keep the same workflow request rate from the micro benchmarks.

Figure 12 shows improvement under the partition-aggregation model with different numbers of RPC servers involved. Bobtail brings improvement of 50% to 65% at the 99th percentile with 10 servers. Similarly to the sequential workloads, the improvement at the 99.9th percentile is relatively small. In addition, as predicted by Figure 7, the reduction in tail completion time diminishes as the number of servers involved in the workload increases. To fully understand this phenomenon, we need to compare the behaviors of these two workload models.

For sequential workloads with random worker assignment, a small number of long-tail nodes have a modest impact. Intuitively, each such node has a $1/N$ chance of being selected for any work item and may (or may not) exhibit long tail behavior for that item. However, when this does happen, the impact is non-trivial, as the long delays consume the equivalent of many "regular" response times. So, one must minimize the pool of long-tail nodes in such architectures but needs not to avoid them entirely.

The situation is less pleasant for parallel, scatter-gather style workloads. In such workloads, long-tail nodes act as the barrier to scalability. Even a relatively low percentage of long-tail nodes will cause significant slowdowns overall, as each phase of the computation runs at the speed of the slowest worker. Reducing or even eliminating long-tail nodes removes this early barrier to scale. However, it is not a panacea. As the computation fans out to more nodes, other limiting factors come into play, reducing the effectiveness of further parallelization. We leave it as future work to study other factors that cause the latency tail problem with larger fan-out in cloud data centers.

6 Discussion

Emergent partitions A naive interpretation of Bobtail's design is that a given customer of EC2 simply seeks out those nodes which have at most one VM per CPU. If this were the case, deploying Bobtail widely would result in a race to the bottom. However, not all forms of sharing are bad. Co-locating multiple VMs running latency-sensitive workloads would not give rise to the scheduling anomaly at the root of our problem. Indeed, wide deployment of Bobtail for latency-sensitive jobs would lead to placements on nodes which are either under-subscribed or dominated by other latency-sensitive workloads. Surprisingly, this provides value to CPU-bound workloads as well. Latency-sensitive workloads will cause frequent context switches and reductions in cache efficiency; both of these degrade CPU-bound workload performance. Therefore, as the usage of Bobtail increases in a cloud data center, we expect it will eventually result in emergent partitions: regions with mostly CPU-bound VMs and regions with mostly latency-sensitive VMs. However, to validate this hypothesis, we would need direct access to the low-level workload characterization of cloud data centers like EC2.

Alternative solutions Bobtail provides a user-centric solution that cloud users can apply today to avoid long latency tails without changing any of the underlying infrastructure. Alternatively, cloud providers can offer their solutions by modifying the cloud infrastructure and placement policy. For example, they can avoid allocating more than *C* VMs on a physical machine with *C* processors, at the cost of resource utilization. They can also overhaul their VM placement policy to allocate different types of VMs in different regions in the first place. In addition, new versions of the credit scheduler [28] may also help alleviate the problem.

7 Related Work

Latency in the Long Tail Proposals to reduce network latency in data centers fall into two broad categories: those that reduce network congestion and those that prioritize flows according to their latency sensitivity. Alizadeh *et al.* proposed to reduce switch buffer occupancy time by leveraging Explicit Congestion Notification (ECN) to indicate the degree of network congestion rather than whether congestion exists [1]. Follow-up work further reduced the buffer occupancy time by slightly capping bandwidth capacity [2]. Wilson *et al.* and Vamanan *et al.* both argued that the TCP congestion control protocols used in data centers should be deadline-aware [26, 22]. Hong *et al.* designed a flow scheduling system for data centers to prioritize latency-sensitive jobs with flow preemption [10]. Zats *et al.* proposed a cross-stack solution that combined ECN with application-specified flow priorities and adaptive load balancing in an effort to unify otherwise disparate knowledge about the state of network traffic [29].

The above solutions focus on the component of long tail flow completion times that is the result of the network alone and, as such, are complementary to our approach. We have shown that the co-scheduling of CPU-intensive and latency-sensitive workloads in virtualized data centers can result in a significant increase in the size of the long tail, and that this component of the tail can be addressed independently of the network.

The Xen Hypervisior and its Scheduler In § 3, we discussed how Xen uses a credit-based scheduler [28] that is not friendly to latency-sensitive workloads. Various characteristics of this credit scheduler have been examined, including scheduler configurations [15], performance interference caused by different types of colocating workloads [15, 12, 25], and the source of overhead incurred by virtualization on the network layer [25]. Several designs have been proposed to improve the current credit scheduler, and they all share the approach of boosting the priority of latency-sensitive VMs while still maintaining CPU fairness in the long term [9, 8, 11]. However, the degree to which such approaches will impact the long tail problem at scale has yet to be studied.

Instead of improving the VM scheduler itself, Wood *et al.* created a framework for the automatic migration of virtual machines between physical hosts in Xen when resources become a bottleneck [27]. Mei *et al.* also pointed out that a strategic co-placement of different workload types in a virtualized data center will improve performance for both cloud consumers and cloud providers [14]. Our work adopts a similar goal of improving the tail completion time of latency-sensitive workloads for individual users while also increasing the overall efficiency of resource usage across the entire virtualized data center. However, our solution does not require the collaboration of cloud providers, and many cloud customers can deploy our system independently.

EC2 Measurements Wang *et al.* showed that the network performance of EC2 is much more variable than that of non-virtualized clusters due to virtualization and processor sharing [24]. In addition, Schad *et al.* found a bimodal performance distribution with high variance for most of their metrics related to CPU, disk I/O, and network [19]. Barker *et al.* also quantified the jitter of CPU, disk, and network performance in EC2 and its impact on latency-sensitive applications [4]. Moreover, A. Li *et al.* compared multiple cloud providers, including EC2, using many types of workloads and claimed that there is no single winner on all metrics [13]. These studies only investigate the average and variance of their performance metrics, while the focus of our study is on the tail of network latency distributions in EC2.

Ou *et al.* considered hardware heterogeneity within EC2, and they noted that within a single instance type and availability zone, the variation in performance for CPU-intensive workloads can be as high as 60% [16]. They made clear that one easy way to improve instance performance is to check the model of processor assigned. While selecting instances also represents the core of our work, Bobtail examines dynamic properties of EC2 as opposed to static configuration properties.

8 Conclusion

In this paper, we demonstrate that virtualization used in EC2 exacerbates the long tail problem of network round-trip-times by a factor of two to four. Notably, we find that poor response times in the cloud are a property of nodes rather than the network, and that the long latency tail problem is pervasive throughout EC2 and persistent over time. Using controlled experiments, we show that co-scheduling of CPU-bound and latency-sensitive tasks causes this problem. We present a system, Bobtail, which proactively detects and avoids these bad neighboring VMs without significantly penalizing node instantiation. Evaluations in two availability zones in EC2's US east region show that common communication patterns benefit from reductions of up to 40% in their 99.9th percentile response times.

9 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, George Porter, for their comments on this paper. This work was supported in part by the Department of Homeland Security (DHS) under contract numbers D08PC75388, and FA8750-12-2-0314, the National Science Foundation (NSF) under contract numbers CNS 1111699, CNS 091639, CNS 08311174, and CNS 0751116, and the Department of the Navy under contract N000.14-09-1-1042.

References

[1] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM'10)* (New Delhi, India, August 2010).

[2] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)* (San Jose, CA, USA, April 2012).

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, October 2003).

[4] Barker, S. K., and Shenoy, P. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM conference on Multimedia systems (MMSys'10)* (Scottsdale, AZ, USA, February 2010).

[5] Bouch, A., Kuchinsky, A., and Bhatti, N. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'00)* (The Hague, The Netherlands, April 2000).

[6] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)* (San Francisco, CA, USA, March 2004).

[7] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Stevenson, WA, USA, October 2007).

[8] Dunlap, G. W. Scheduler Development Update. In *Xen Summit Asia 2009* (Shanghai, China, November 2009).

[9] Govindan, S., Nath, A. R., Das, A., Urgaonkar, B., and Sivasubramaniam, A. Xen and Co.: Communication-Aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE'07)* (San Diego, CA, 2007, June 2007).

[10] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).

[11] Kim, H., Lim, H., Jeong, J., Jo, H., and Lee, J. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 5th international conference on virtual execution environments (VEE'09)* (Washington, DC, USA, March 2009).

[12] Koh, Y., Knauerhase, R. C., Brett, P., Bowman, M., Wen, Z., and Pu, C. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'07)* (San Jose, CA, USA, April 2007).

[13] Li, A., Yang, X., Kandula, S., and Zhang, M. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)* (Melbourne, Australia, November 2010).

[14] Mei, Y., Liu, L., Pu, X., and Sivathanu, S. Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD'10)* (Miami, FL, June 2010).

[15] Ongaro, D., Cox, A. L., and Rixner, S. Scheduling I/O in virtual machine monitors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)* (Washington, DC, USA, March 2008).

[16] Ou, Z., Zhuang, H., Nurminen, J. K., Ylä-Jääski, A., and Hui, P. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)* (Boston, MA, USA, June 2012).

[17] Patterson, D. A. Latency lags bandwidth. *Communication of ACM 47*, 10 (Oct 2004), 71–75.

[18] Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)* (Napa, CA, USA, May 2011).

[19] Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)* (Singapore, September 2010).

[20] Slee, M., Agarwal, A., and Kwiatkowski, M. Thrift: Scalable Cross-Language Services Implementation. Tech. rep., Facebook, Palo Alto, CA, USA, April 2007.

[21] Techcrunch. There Goes The Weekend! Pinterest, Instagram And Netflix Down Due To AWS Outage. http://techcrunch.com/2012/06/30/there-goes-the-weekend-pinterest-instagram-and-netflix-down-due-to-aws-outage/.

[22] Vamanan, B., Hasan, J., and Vijaykumar, T. N. Deadline-Aware Datacenter TCP (D^2TCP). In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).

[23] Vasudevan, V., Phanishayee, A., Shah, H., Krevat, E., Andersen, D. G., Ganger, G. R., Gibson, G. A., and Mueller, B. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 conference (SIGCOMM'09)* (Barcelona, Spain, August 2009).

[24] Wang, G., and Ng, T. S. E. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th conference on Information communications (INFOCOM'10)* (San Diego, CA, USA, March 2010).

[25] Whiteaker, J., Schneider, F., and Teixeira, R. Explaining Packet Delays Under Virtualization. *SIGCOMM Computer Communication Review 41*, 1 (January 2011), 38–44.

[26] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM'11)* (Toronto, ON, CA, August 2011).

[27] Wood, T., Shenoy, P. J., Venkataramani, A., and Yousif, M. S. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th conference on Symposium on Networked Systems Design & Implementation (NSDI'07)* (Cambridge, MA, USA, April 2007).

[28] xen.org. Xen Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`.

[29] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)* (Helsinki, Finland, August 2012).