Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense)

Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley,
Thomas Ristenpart and Michael M. Swift
University of Wisconsin–Madison
{venkatv,farleyb,rist,swift}@cs.wisc.edu, thawan@fb.com

ABSTRACT

Cloud computing promises great efficiencies by multiplexing resources among disparate customers. For example, Amazon's Elastic Compute Cloud (EC2), Microsoft Azure, Google's Compute Engine, and Rackspace Hosting all offer Infrastructure as a Service (IaaS) solutions that pack multiple customer virtual machines (VMs) onto the same physical server.

The gained efficiencies have some cost: past work has shown that the performance of one customer's VM can suffer due to interference from another. In experiments on a local testbed, we found that the performance of a cache-sensitive benchmark can degrade by more than 80% because of interference from another VM.

This interference incentivizes a new class of attacks, that we call *resource-freeing attacks* (RFAs). The goal is to modify the workload of a victim VM in a way that frees up resources for the attacker's VM. We explore in depth a particular example of an RFA. Counter-intuitively, by adding load to a co-resident victim, the attack speeds up a class of cache-bound workloads. In a controlled lab setting we show that this can improve performance of synthetic benchmarks by up to 60% over not running the attack. In the noisier setting of Amazon's EC2, we still show improvements of up to 13%.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection; D.4.1 [**Operating System**]: Process Management—*scheduling*; K.6.5 [**Management of Computing and Information System**]: Security and Protection—*physical security*

General Terms

Economics, Experimentation, Measurement, Performance, Security

Keywords

cloud computing, virtualization, scheduling, security, resource-freeing attacks

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.
Copyright 2012 ACM 978-1-4503-1651-4/12/10 ...$15.00.

1. INTRODUCTION

Cloud computing provides high efficiency in part by multiplexing multiple customer workloads onto a single physical machine. For example, Amazon's Elastic Compute Cloud (EC2) [3] runs multiple customer virtual machines (VMs) on a single host. For small instances, they offer each guest VM roughly 40% of a single CPU by time slicing. Similarly, access to the local disk, network, memory, and cache are all shared by virtual machines from multiple customers.

However, with this efficiency comes performance interference. When two customer applications share a machine, they contend for access to resources. Existing hardware and software virtualization mechanisms do not provide perfect performance isolation. For example, running two applications that make heavy use of memory bandwidth can degrade the performance of both. Past work has demonstrated the existence and amount of this interference [5, 19].

As a result, there have been numerous proposals on how to construct hypervisors or processors that better isolate customer applications from each other. For example, fine-grained accounting of CPU usage [12], network traffic [28] or disk-queue utilization [11] can decrease the amount of interference. Unfortunately, the inherent tension between efficiency and isolation means that, in practice, cloud computing systems continue to provide poor isolation. In experiments on a local Xen [4] testbed, we show, for example, that certain cache-sensitive workloads take 5x longer when contending with other memory-intensive workloads.

Unlike in private data centers, such contention in public clouds arises between disparate customers. Unique to the public cloud setting, then, is the incentive for greedy customers to attempt to free up resources for their application by interfering with other customers' use of them. A clear-cut example would be a malicious customer crashing co-resident VMs, but this requires knowledge of an exploitable vulnerability and would be easily detectable. We are interested in whether there exist more subtle strategies for freeing up resources.

We explore an approach based on two observations. First, applications are often limited by a single bottleneck resource, such as memory or network bandwidth. Second, we observe that an application's use of resources can change unevenly based on workload. For example, a web server may be network limited when serving static content, but CPU limited when serving dynamic content.

A *resource-freeing attack* (RFA) leverages these observations to improve a VM's performance by forcing a competing VM to saturate some bottleneck. If done carefully, this can slow down or shift the competing application's use of a desired resource. For example, we investigate in detail an RFA that improves cache performance when co-resident with a heavily used Apache web server. Greedy users will benefit from running the RFA, and the victim ends up paying for increased load and the costs of reduced legitimate traffic.

We begin this paper with a comprehensive study of the resource interference exhibited by the Xen hypervisor in our local testbed. In addition to testing for contention of a single resource, these results show that workloads using different resources can contend as well, and that scheduling choices on multicore processors greatly affect the performance loss. We then develop a proof-of-concept resource-freeing attack for the cache-network contention scenario described above. In a controlled environment, we determine the necessary conditions for a successful resource-freeing attack, and show that average performance of a cache-sensitive benchmark can be improved by as much as 212% when the two VMs always share a single core, highlighting the potential for RFAs to ease cache contention for the attacker. If VMs float among all cores (the default configuration in Xen), we still see performance gains of up to 60%. When applied to several SPEC benchmarks [13], whose more balanced workloads are less effected by cache contention, RFAs still provide benefit: in one case it reduces the effect of contention by 66.5% which translated to a 6% performance improvement.

Finally, we show that resource-freeing attacks are possible in uncontrolled settings by demonstrating their use on Amazon's EC2. Using co-resident virtual machines launched under accounts we control, we show that introducing additional workload on one virtual machine can improve the performance of our cache-sensitive benchmark by up to 13% and provides speedups for several SPEC benchmarks as well.

2. SCHEDULING AND ISOLATION IN VIRTUALIZED ENVIRONMENTS

A key reason to use hypervisors in cloud computing is their ability to provide performance isolation between customer applications. Indeed, isolation was a primary goal for the original development of the Xen hypervisor used in EC2 [4]. Perfect performance isolation should guarantee that the behavior of one guest virtual machine does not affect the performance of other guest virtual machines. To this end, Xen and other virtual machine monitors (VMMs) focus on fairly allocating CPU time and memory capacity [33]. However, other hardware resources such as memory bandwidth, cache capacity, network, and disk have received less attention.

In order to understand the sources and effects of performance interference, we describe the Xen hypervisor mechanisms and policies for sharing resources between guest virtual machines while still providing performance isolation.

CPU. The Xen scheduler provides both fair-share allocation of CPU and low-latency dispatch for I/O-intensive VMs. We use the credit scheduler [7] in our experiments, as it is most commonly used in deployments. The scheduler views VMs as a set of virtual CPUs (VCPUs), and its task is to determine which VCPUs should be run on each physical CPU at any given time.

The scheduler gives VCPUs *credits* at a pre-determined rate. The credits represent a share of the CPU and provide access to the CPU. Every 10 ms a periodic scheduler tick removes credits from the currently running VCPU and if it has none remaining, switches to the next VCPU in the ready queue. VCPUs are given more credits periodically (typically every 30 ms). Thus, if a CPU-bound process runs out of credit, it must suspend for up to 30 ms until it receives new credits to run. A process that runs for short periods may never run out of credit, although the total amount it can accrue is limited.

In order to support low-latency I/O, Xen implements a *boost* mechanism that raises the priority of a VM when it receives an interrupt, which moves it towards the head of the ready queue. This allows it to preempt the running VM and respond to an I/O request immediately. However, a VM that has run out of credits cannot receive boost. The boost mechanism is a key component of the resource-freeing attack we introduce in Section 5.

The credit scheduler supports a work-conserving mode, in which idle CPU time is distributed to runnable VMs, and a non-work-conserving mode, in which VMs' CPU time is capped. The latter mode reduces efficiency but improves performance isolation. Though Amazon does not report which mode it uses, our experiments indicate that EC2 uses non-work-conserving scheduling.

On a multiprocessor, Xen can either *float* VPCUs, letting them execute on any CPU, or *pin* them to particular CPUs. When floating, Xen allows a VCPU to run on any CPU *unless* it ran in the last 1 ms, in which case it is rescheduled on the same core to maintain cache locality. We determined experimentally that EC2 allows VCPUs to float across cores.

Memory. Xen isolates memory access primarily by controlling the allocation of memory pages to VMs. In cloud settings, Xen is often configured to give each VM a static number of pages. It does not swap pages to disk, actively manage the amount of memory available to each VM, or use deduplication to maximize use of memory [33]. Furthermore, x86 hardware does not provide the ability to enforce per-VCPU limits on memory bandwidth or cache usage. Hence, these are not managed by Xen.

Devices. By default, Xen seeks fair sharing of disk and network by processing batches of requests from VMs in round-robin order [4]. For disks, this can lead to widely varying access times, as sets of random requests may incur a longer delay than sequential accesses. The Xen default is to make device scheduling work conserving, so performance can also degrade if another VM that was not using a device suddenly begins to do so. However, we observe that EC2 sets caps on the network bandwidth available to an m1.small instance at around 300 Mbps, but does not cap disk bandwidth.

3. RESOURCE-FREEING ATTACKS

The interference encountered between VMs on public clouds motivates a new class of attacks, which we call resource-freeing attacks (RFAs). The general idea of an RFA is that when a guest virtual machine suffers due to performance interference, it can affect the workload of other VMs on the same physical server in a way that improves its own performance.

Attack setting. We consider a setting in which an *attacker* VM and one or more *victim* VMs are co-resident on the same physical server in a public cloud. There may be additional co-resident VMs as well. It is well known that public clouds make extensive use of multi-tenancy.

The RFAs we consider in Section 5 assume that the victim is running a public network service, such as a web server. This is a frequent occurrence in public clouds. Measurements in 2009 showed that approximately 25% of IP addresses in one portion of EC2's address space hosted a publicly accessible web server [25].

Launching RFAs that exploit a public network service require that the attacker knows with whom it is co-resident. On many clouds this is straightforward: the attacker can scan nearby internal IP addresses on appropriate ports to see if there exist public network services. This was shown to work in Amazon EC2, where for example m1.small co-resident instances had internal IP addresses whose numerical distance from an attacker's internal IP address was at most eight [25]. Furthermore, packet round-trip times can be used to verify co-residence. We expect that similar techniques work on other clouds, such as Rackspace.

The attacker seeks to interfere with the victim(s) to ease contention for resources on the node or nearby network. The attacker consists of two logical components, a *beneficiary* and a *helper*. The beneficiary is the application whose efficiency the attacker seeks to improve. The helper is a process, either running from within the same instance or on another machine, that the attacker will use to introduce new workload on the victim. Without loss of generality, we will describe attacks in terms of one victim, one beneficiary, and one helper.

We assume the beneficiary's performance is reduced because of interference on a single contended resource, termed the *target resource*. For example, a disk-bound beneficiary may suffer from competing disk accesses from victim VMs.

Conceptual framework. The beneficiary and the helper work together to change the victim's resource consumption in a manner that frees up the target resource. This is done by increasing the time the victim spends on one portion of its workload, which limits its use of other resources.

There are two requirements for an RFA. First, an RFA must raise the victim's usage of one resource until it reaches a *bottleneck*. Once in a bottleneck, the victim cannot increase usage of any resources because of the bottleneck. For example, once a web server saturates the network, it cannot use any more CPU or disk bandwidth. However, simply raising the victim to a bottleneck does not free resources; it just prevents additional use of them. The second requirement of an RFA is to *shift* the victim's resource usage so that a greater fraction of time is spent on the bottleneck resource, which prevents spending time on other resources. Thus, the bottleneck resource crowds out other resource usage. As an example, a web server may be sent requests for low-popularity web pages that cause random disk accesses. The latency of these requests may crowd requests for popular pages and overall reduce the CPU usage of the server.

There are two shifts in target resource usage that can help the beneficiary. First, if the victim is forced to use less of the resource, then there may be more available for the beneficiary. Second, even if the victim uses the same amount of the resource, the accesses may be shifted in time. For example, shifting a victim's workload so that cache accesses are consolidated into fewer, longer periods can aid the beneficiary by ensuring it retains cache contents for a larger percentage of its run time. A similar effect could be achieved for resources like the hard disk if we are able to provide the beneficiary with longer periods of uninterrupted sequential accesses.

Modifying resource consumption. The helper modifies the victim's resource usage and pushes it to overload a bottleneck resource. This can be done externally, by introducing new work over the network, or internally, by increasing contention for other shared resources.

A helper may introduce additional load to a server that both increases its total load and skews its workload towards a particular resource. The example above of requesting unpopular content skews a web server's resource usage away from the CPU towards the disk. This can create a bottleneck at either the server's connection limit or disk bandwidth. Similarly, the helper may submit CPU-intensive requests for dynamic data that drive up the server's CPU usage until it exceeds its credit limit and is preempted by the hypervisor.

The helper can also affect performance by increasing the load on other contended resources. Consider again a web server that makes use of the disk to fetch content. A helper running in the beneficiary's instance can introduce unnecessary disk requests in order to degrade the victim's disk performance and cause the disk to become a bottleneck. Similarly, the helper could slow the victim by introducing additional network traffic that makes network bandwidth a bottleneck for the server.

Xen Version	4.1.1
Xen Scheduler	Credit Scheduler 1
OS	Fedora 15, Linux 2.6.40.6-0.fc15
Dom0	4 VCPU / 6 GB memory / no cap / weight 512
DomU	8 instances each with 1 VCPU / 1 GB memory / 40% cap / weight 256
Network	Bridging via *Dom0*
Disk	5 GB LVM disk partition of a single large disk separated by 150GB

Figure 1: Xen configuration in our local testbed.

There exist some obvious ways an attacker might modify the workload of a victim. If the attacker knows how to remotely crash the victim via some exploitable vulnerability, then the helper can quite directly free up the target resource (among others). However this is not only noisy, but requires a known vulnerability. Instead, we focus on the case that the attacker can affect the victim only through use (or abuse) of legitimate APIs.

Example RFA. As a simple example of an RFA, we look at the setting of two web servers, running in separate VMs on the same physical node, that compete for network bandwidth. Assume they both serve a mix of static and dynamic content. Under similar loads, a work-conserving network scheduler will fairly share network capacity and give each web server 50% (indeed, our experiment show that Xen does fairly share network bandwidth).

However, if we introduce CPU-intensive requests for dynamic content to one web server that saturate the CPU time available to the server, we find that the other server's share of the network increases from 50% to 85%, because there is now less competing traffic. We note that this requires a work-conserving scheduler that splits excess network capacity across the VMs requesting it. A non-work conserving scheduler would cap the bandwidth available to each VM, and thus a decline in the use by one VM would not increase the bandwidth available to others.

4. CONTENTION MEASUREMENTS

In order to understand which resources are amenable to resource-freeing attacks in a Xen environment, we created a local testbed that attempts to duplicate a typical configuration found in EC2 (in particular, the m1.small instance type).

Testbed. Although Amazon does not make their precise hardware configurations public, we can still gain some insight into the hardware on which an instance is running by looking at system files and the CPUID instruction. Based on this, we use a platform consisting of a 4-core, 2-package 2.66 GHz Intel Xeon E5430 with 6MB of shared L2 cache per package and 4GB of main memory. This is representative of some of the architectures used by EC2.

We install Xen on the testbed, using the configurations shown in Figure 1. Again, while we do not have precise knowledge of Amazon's setup for Xen, our configuration approximates the EC2 m1.small instance.

This configuration allows us to precisely control the workload by varying scheduling policies and by fixing workloads to different cores. In addition, it enables us to obtain internal statistics from Xen, such as traces of scheduling activities.

Figure 2 describes the workloads we use for stressing different hardware resources. The workloads run in a virtual machine with one VCPU. In order to understand the impact of sharing a cache, we execute the workloads in three scenarios:

Workload	Description
CPU	Solving the N-queens problem for $N = 14$.
Net	Lightweight web server hosting 32KB static web pages cached in memory, 5000 requests per second from a separate client.
Diskrand	Requests for randomly selected 4KB chunk in 1 GB span.
Memrand	Randomly request 4B from every 64B of data from a 64MB buffer.
LLC	Execute LLCProbe, which sequentially requests 4B from every 64B of data within an LLC-sized buffer using cache coloring to balance access across cache sets.

Figure 2: Resource-specific workloads used to test contention.

(i) *Same core* time slices two VMs on a single core, which shares all levels of processor cache.

(ii) *Same package* runs two VMs each pinned to a separate core on a single package, which shares only the last-level cache.

(iii) *Different package* runs two VMs floating over cores on different packages, which do not share any cache, but do share bandwidth to memory.

In addition, Xen uses a separate VM named *Dom0* to run device drivers. In accordance with usage guides, we provision *Dom0* with four VCPUs. As past work has shown this VM can cause contention [36, 12], we make it execute on a different package for the first two configurations and allow it to use all four cores (both cores in both packages) for the third.

Extent of Resource Contention. The goal of our experiments is to determine the contention between workloads using different hardware resources and determine whether enough contention exists to mount an RFA. With perfect isolation, performance should remain unchanged no matter what competing benchmarks run. However, if the isolation is not perfect, then we may see performance degradation, and thus may be able to successfully mount an RFA.

Figure 3 provides tables showing the results, which demonstrate that Xen is not able to completely isolate the performance of any resource. Across all three configurations, CPU and Memrand show the least interference, indicating that Xen does a good job accounting for CPU usage and that the processor limits contention for memory bandwidth.

However, for all other resources, there are competing workloads that substantially degrade performance. The two resources suffering the worst contention are Diskrand where run time increases 455% with contending random disk access; and LLC, where run time increases over 500% with Net and over 500% with Memrand. for Diskrand, competing disk traffic causes seeks to be much longer and hence slower. For LLC, competing workloads either interrupt frequently (Net) or move a lot of data through the cache (Memrand).

The three configurations differ mostly in the LLC results. In the same-core and different-package configurations, the contention with LLC is fairly small. On the same core, the conflicting code does not run concurrently, so performance is lost only after a context switch. On different packages, performance losses come largely from *Dom0* , which is spread across all cores. In the same-package configuration, though, the tests execute concurrently and thus one program may displace data while the other is running.

One pair of resources stands out as the worst case across all configurations: the degradation caused by Net on LLC. This occurs for three reasons: (i) the HTTP requests cause frequent interrupts and hence frequent preemptions due to boost; (ii) in the same-core and same-package configurations the web server itself runs frequently and displaces cache contents; and (iii) *Dom0* runs the NIC device driver in the different-package configuration. We will therefore focus our investigation of RFAs on the conflict between such workloads, and leave exploration of RFAs for other workload combinations to future work.

Same core	CPU	Net	Diskrand	Memrand	LLC
CPU	-	5	-	-	-
Net	-	194	-	-	-
Diskrand	-	-	455	-	-
Memrand	-	6	-	-	-
LLC	8	539	72	38	34

Same package	CPU	Net	Diskrand	Memrand	LLC
CPU	-	-	-	-	-
Net	-	198	-	-	-
Diskrand	-	-	461	-	-
Memrand	-	-	17	-	-
LLC	20	448	55	566	566

Diff. package	CPU	Net	Diskrand	Memrand	LLC
CPU	-	20	-	-	-
Net	-	100	-	-	-
Diskrand	-	-	462	-	-
Memrand	-	35	-	-	-
LLC	6	699	11	15	15

Figure 3: Percentage increase in workload run times indicated in row when contending with workload indicated in column. Percentage is computed as run time with contention over run time on otherwise idle machine. For network, run time is the time to serve a fixed number of requests. A dash means there was no significant performance degradation. (Top) The VMs are pinned to the same core. (Middle) The VMs are pinned to different cores on the same package. (Bottom) The VMs are pinned to different packages.

5. RFA FOR CACHE VERSUS NETWORK

As we saw, a particularly egregious performance loss is felt by cache-bound workloads when co-resident with a network server. Unfortunately, co-residence of such workloads seems a likely scenario in public clouds: network servers are a canonical application (EC2 alone hosts several million websites [21]) while cache-bound processes abound. The remainder of the paper seeks to understand whether a greedy customer can mount an RFA to increase performance when co-resident with one or more web servers.

Setting. We start by providing a full description of the setting on which we focus. The beneficiary is a cache bound program running alone in a VM with one VCPU. We use the LLCProbe benchmark as stand-in for a real beneficiary. LLCProbe is intentionally a synthetic benchmark and is designed to expose idealized worst-case behavior. Nevertheless, Its pointer-chasing behavior is reflected in real workloads [2]. We will also investigate more balanced benchmarks such as SPEC CPU2006 [13], SPECjbb2005 [1] and graph500 [2].

In addition to the beneficiary, there is a victim VM co-resident on the same physical machine running the Apache web server (version 2.2.22). It is configured to serve a mix of static and dynamic content. The static content consists of $4,096$ $32KB$ web pages (enough to overlow the 6MB LLC) containing random bytes. The dynamic content is a CGI script that can be configured to consume varying amounts of CPU time via busy looping. This script serves as a stand in for either an actual web server serving dynamic content on the web, or the effects of DoS attacks that drive up CPU usage, such as complexity attacks [8, 9]. The script takes a parameter to control duration of the attack, and spins until wall-clock time advances that duration. We note that this does not reflect the behavior of most DoS attacks, which take a fixed number of cycles, but we use it to provide better control over the web server's behavior. We confirmed that the behaviors exhibited also arise with CGI scripts performing a fixed number of computations.

The Apache server is configured with the *mod_mem_cache* module to reduce the latency of static content and FastCGI to pre-fork a process for CGI scripts. We also use the Multi-Processing Module for workers, which is a hybrid multithreaded multi-process Apache web server design used for better performance and for handling larger request loads.

To simulate load on the web server, we use a custom-built multithreaded *load generator* that sends web requests for the static content hosted by the victim. Each client thread in the load generator randomly selects a static web page to request from the web server. The load generator includes a rate controller thread that ensures that the actual load on the web server does not exceed the specified request rate. The client uses 32 worker threads, which we empirically determined is enough to sustain the web server's maximum rate. Requests are synchronous and hence the load generator waits for the response to the previous request and then a timeout (to prevent sending requests too fast) before sending the next request. Since each thread in the load generator waits for a response from the web server before sending the next request, it may not meet the specified request rate if the server or the network bandwidth cannot sustain the load. The helper, which performs the actual RFA, is identical to the load generator except that it sends requests for the CGI script rather than for static pages.

Understanding the contention. We conduct experiments on our local testbed to understand the basic performance degradation experienced by LLCProbe as the web server's workload varies. We report the average time to probe the cache; one probe involves accessing every cacheline out of a buffer of size equal to the LLC. We measure the time per probe by counting the number of probes completed in 10 seconds.

To understand contention, we first pin the victim VM and the beneficiary VM to the same core and pin *Dom0* to a different package. The *Fixed Core* columns in Figure 4 show the runtime per cache probe averaged over 3 runs for a range of background request rates to the web sever. The *Perf. Degradation* column shows the percent increase in probe time relative to running with an idle victim VM.

Request	Fixed Core		Floating Core	
Rate	Runtime	Increase	Runtime	Increase
0	4033	0	4791	0
100	4780	19%	5362	12%
1000	6500	61%	6887	44%
1500	7740	92%	7759	62%
2000	9569	137%	8508	78%
3000	18392	356%	16630	247%

Figure 4: Runtimes (in microseconds) and percentage increase of LLCProbe (foreground) workload as a function of request rate to victim (background). For *Fixed Core* both VMs are pinned to the same core and for *Floating Core* Xen chooses where to execute them.

As the workload of the victim increases, we see a corresponding increase in the performance degradation of LLCProbe. To evaluate our hypothesis that the effect arises due to frequent interruptions, we use Xentrace [17] to record the domain switches that occur over a fixed period of time in which the LLCProbe VM runs. We analyzed the case of 1500 requests per second (rps) and 3000 rps. For the 3000 rps case, the web server runs for less than 1 ms in 80% of the times it is scheduled whereas in the 1500 rps case the web server runs for less than 1 ms only 40% of the time, because the longer run periods reflect fixed-length CPU tasks not correlated with traffic. Because Apache does not saturate its CPU allocation, it retains "boost" priority, which allows it to preempt LLCProbe for every request. Thus, LLCProbe also runs for short periods, causing it to lose the data in its cache.

The rightmost columns in Figure 4 show the same experiment when the two VMs are allowed to float across all the cores (*floating*). We see a similar trend here, though slightly less severe because for some fraction of time, the victim and beneficiary VMs are scheduled on different packages and do not share an LLC. Thus, we expect in live settings such as EC2 to see less interference than when both VMs are pinned to the same core.

We separately investigate the effect of contention with the Xen driver domain, *Dom0* [1], which handles all device access such as interrupts or requests to send a packet. In the typical setting where *Dom0* is assigned one VCPU per physical CPU, *Dom0* may run on any core and uses the same scheduling mechanism as other guest VMs. As a result, *Dom0* receives boost and can interfere with the beneficiary just like the victim when it handles a network interrupt. *Dom0* and the beneficiary may share a CPU even if the victim is scheduled elsewhere.

The attack. As alluded to in Section 4, the beneficiary's performance degradation is caused by a victim frequently preempting the beneficiary and thereby polluting its cache. The preemptions occur to handle static web page requests due to legitimate traffic to the victim. Our attack aims to exploit the victim's CPU allotment as a bottleneck resource in order to shift, in time, its accesses to the cache, and to reduce the number of requests it serves. Doing so will provide the beneficiary longer periods of uninterrupted access to the cache and less cache pollution from handling requests, resulting in increased cache hit rates and improved performance.

The trigger for this is the introduction of a small number of CGI requests per second from a helper. Even a low rate of requests per second can push the victim up to its CPU cap, forcing it to lose boost and thus consolidating its use of the cache into a smaller time frame. Introducing long-latency dynamic requests means that, instead of interrupting LLCProbe frequently, the web server runs continuously until the Xen scheduler preempts it, which allows LLCProbe to run uninterrupted. The Xen credit scheduler allows a maximum of 30 ms of credit per VCPU, with each domain being allotted only one VCPU in our case. Therefore, the helper sends *RFA requests* that invoke the CPU-intensive CGI helper in an effort to use up the victim's CPU allotment. In addition, the CPU-intensive requests displace legitimate traffic and thus reduce the rate of requests that pollute the cache.

Here the helper is any system that can make CGI requests. Given the very low rate required, this could be a free micro instance running on the cloud or —scaling up— a single system that performs the RFA against many victims in parallel (that are each co-resident with a different beneficiary). While for some applications the helper might be put to better use helping with whatever computation the beneficiary is performing, in others this will not be possible (e.g., if it is not easily parallelized) or not as cost effective. We also mention that one might include a lightweight helper on the same VM as

[1] The default configuration in Xen is to run device drivers in a single domain with privileged access to I/O hardware.

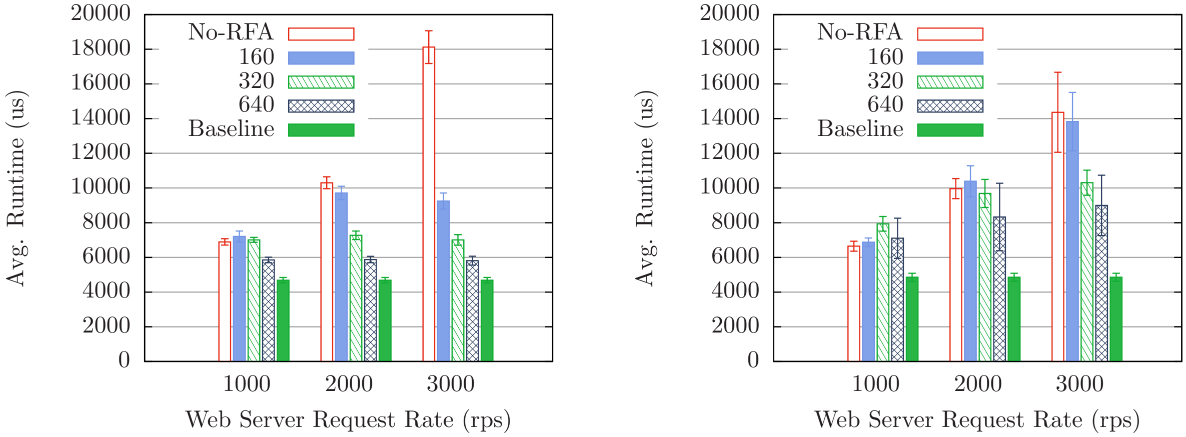


Figure 5: Performance of LLCProbe workload when pinned to same core as co-resident web server. "Baseline" measures baseline performance when no traffic was sent to the victim; it is shown in each grouping for comparison. "No-RFA" measures performance when no RFA requests were sent. (Left) Performance when LLCProbe and web server VMs are pinned to same core. (Right) Performance when they float amongst cores. Error bars indicate one standard deviation.

the beneficiary, but this would require care to ensure that interference from the client does not outweigh the potential speedup due to the RFA. In our experiments to follow, we run the helper on a system different from the one on which the beneficiary and victim co-reside.

5.1 Evaluation on Local Testbed

The results above show that LLCProbe experiences a significant performance gap when running on an otherwise idle server as opposed to one that is hosting one or more active web servers. In this section, we show that this performance gap can be narrowed using the RFA outlined above. In the following we look at the effectiveness of the attack under a range of *RFA intensities*, which specifies the their total runtime per second. Unless otherwise noted, we implement the RFA using CGI requests specifying 40 ms of computation. We investigate a range of RFA intensities: 160, 320, and 640 ms. This allows understanding both the effect of overloading the victim by requesting more computation than its total allotment of 400 ms.

We first run LLCProbe fifteen times while the victim VM is idle to get a baseline. Then for each legitimate victim traffic rate and each level of RFA including "No-RFA", we run LLCProbe fifteen times while offering the appropriate legitimate traffic and RFA traffic.

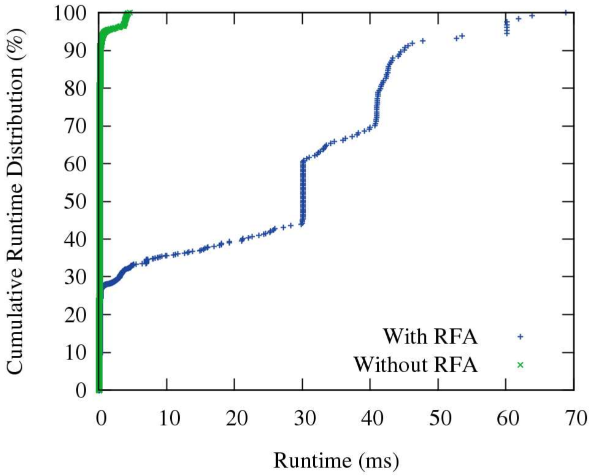
The average runtimes of these tests are shown in Figure 5. We observe several interesting trends. Consider the left chart, which reports on a setting with both victim and beneficiary pinned to the same core and all four *Dom0* VCPUs floating across all cores. First, introducing the extra load from the RFA requests helps the beneficiary. Second, the greater the victim's load the higher the payoffs from the RFA.

In order to understand these results, we ran additional experiments trying to identify various sources of interference on the beneficiary. There are three main sources of interference: two effects on request processing by the web server and the effect of network packet processing by *Dom0* . RFA requests help mitigate the effect of web server request handling in two ways. First, introducing sufficiently many CPU-intensive requests will deprive the web server of the boost priority. This is the major reason for the high performance improvement in the pinned case shown in Figure 5. Second, introducing long-running CGI requests reduces the amount of CPU time available to serve legitimate traffic and thus, implicitly reduces the capacity of the web server. This is the reason for higher payoffs at higher web-server request rates. Reducing *Dom0* 's impact on the beneficiary can only be indirectly achieved by saturating the web server and hence reducing the rate of incoming request to the web server.

Figure 6 shows the CDF of runtime durations of the web server (top chart) and LLCProbe (bottom chart) before being preempted both with and without an RFA for the pinned case. What we see is that LLCProbe runs for more than 1 ms 85% of the time in the RFA case but only 60% of the time without the RFA. This accounts for part of its improved performance. Similarly, the web server changes from running longer than 1 ms for only 10% of the time to 60% of the time. Furthermore, we can see that the web server often runs out of scheduling credit from the vertical line at 30 ms, indicating that it uses up some of its scheduling quanta.

Figure 7 shows the effect of displacing legitimate traffic at higher RFA intensities for the floating case. At low web-server request rates and low RFA intensities, the offered and the observed load remain similar. However, at 3000 rps and RFA intensity of 320, the observed load reduces to 1995 rps, which leads LLCProbe to have performance similar to No-RFA case at 2000 rps (right graph in Figure 5). This is the primary reason for large performance improvement at 3000 rps in both pinned and floating case shown in Figure 5.

In the floating case shown on the right in Figure 5, we see that RFA requests can sometimes hurt performance. There appear to be two reasons for this. First, some percentage of the time LLCProbe and Apache are running concurrently on two different cores sharing an LLC. Because the two loads run concurrently, every cache access by the web server hurts the performance of LLCProbe. In such a case, depriving the web server of boost is insufficient and LLCProbe performance increases only when the RFA rate is high enough so that the web server saturates its CPU allotment and so spends more than half the time waiting (40% CPU cap). In a separate experiment, we pinned the web server and the LLCProbe to different cores on the same package, and used a web-server request rate of 2000 rps. In this configuration, a high RFA intensity improved performance by a meager 2.4%. In contrast, when we pin the two to the same core, performance improved by 70%. Thus, improving performance when sharing a core is possible without re-



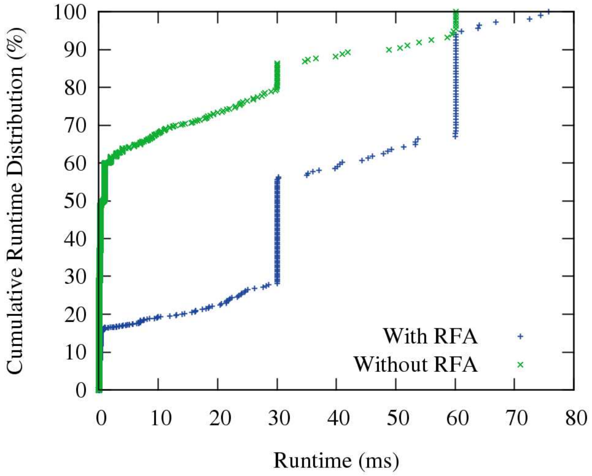


Figure 6: Cumulative runtime distribution of (top) the web server domain (with load 2,000 rps) and (bottom) the LLCProbe domain under both no RFA and with RFA 320 in pinned core case.

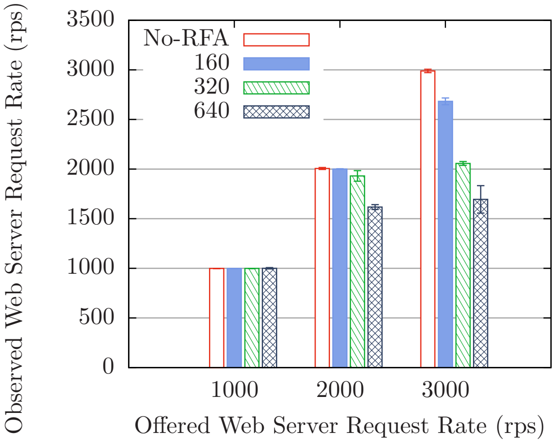


Figure 7: Offered vs. observed load on web server with varying RFA intensities when all the VMs float across all cores.

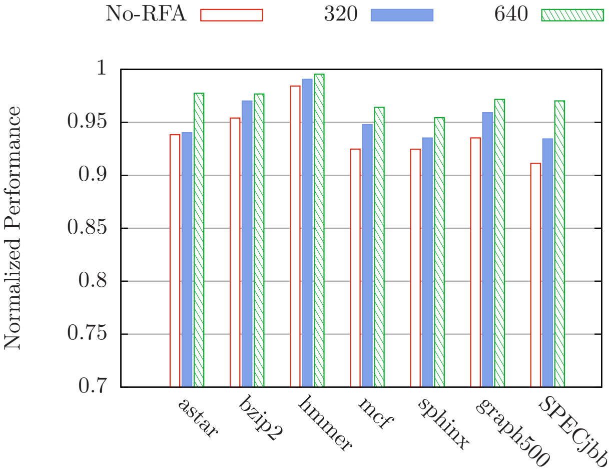


Figure 8: Normalized performance (baseline runtime over runtime) for SPEC workloads on our local testbed for various RFA intensities. All values are at a web server request rate of 3000 rps.

ducing legitimate foreground traffic, while without sharing a core it requires displacing some legitimate traffic.

Second, in this floating case the beneficiary will for some percentage of the time be scheduled to run on a core or package as *Dom0* . Since *Dom0* handles all incoming and outgoing packets, it may frequently interrupt the beneficiary and pollute its cache state. When we pin LLCProbe and the web server to different packages (no shared cache) but let *Dom0* float, LLCProbe still experiences interference. At a load of 2000 rps on the web server, LLCProbe suffered a 78% degradation in performance just due to *Dom0* 's inference. The RFA we explore can only alleviate contention from *Dom0* by forcing a drop in the web server's foreground traffic rate (by exhausting its VM's CPU allocation as shown in Figure 7).

Finally, we analyze a spectrum of SPEC benchmarks. Each SPEC benchmark is run three times with an idle webserver, an active web server, and an active web server with various RFA intensities where all the VMs (including *Dom0*) float across all cores. Figure 8 depicts the normalized performance of seven benchmarks under no RFA and intensities of 320 and 640. That is, the reported fractions are computed as t'/t where t is the average runtime (request latency is computed and used for SPECjbb) and t' is the average baseline performance when no traffic is sent to the victim. All benchmarks benefit from the RFA, with the general trend that cache-sensitive benchmarks (as indicated by a larger drop in performance relative to the baseline) achieve more gains from the RFA. For example, the 640 RFA increases normalized performance of SPECjbb from 0.91 to 0.97, a 6 percentage point improvement in performance and a 66.5% reduction in harm due to contention. The smallest improvement occurs with hmmer, which shows only a 1.1 percentage point improvement because it only suffers a performance loss of 1.6% without the RFA. Across all the benchmarks, the 640 RFA achieves an average performance improvement of 3.4 percentage points and recovers 55.5% of lost performance. These improvements come largely from the ability of the RFA to reduce the request rate of the victim web server.

5.2 Evaluation on EC2

The above experiments clearly indicate that RFAs can provide substantial gains in a controlled setting. To verify that the attacks will also work in a noisier, more realistic setting, we turn to Amazon's Elastic Compute Cloud (EC2). There are several reasons it is important to evaluate RFAs in a real cloud setting. First of all, the success of the RFA is highly dependent on the overall load of the physical machine. The instances in question (the beneficiary and the victim) make up only a portion of the total possible load on a single machine. If the other instances on the machine are heavy resource users, they will constantly interfere with the beneficiary and overshadow any performance benefit from slowing the victim. Thus, if most physical machines in EC2 are constantly under heavy load, we are unlikely to see much effect from an RFA on a single victim. Furthermore, EC2's Xen configuration is not publicly available and may prevent RFAs. Thus, to understand if RFAs actually behave as an attacker would hope, it is necessary to verify their effectiveness in a live setting like EC2.

Ethical considerations. When using EC2 for experiments, we are obligated to consider the ethical, contractual, and legal implications of our work. In our experiments, we use instances running under our accounts in our names as stand-ins for RFA victims and beneficiaries. We abide by the Amazon user agreement, and use only the legitimate Amazon-provided APIs. We only attempt to send reasonable levels of traffic (slightly more than 2000 rps for a small web page) to our own instances (the stand-ins for victims). We do not directly interact with any other customer's instances. Our experiments are therefore within the scope of typical customer behavior on EC2: running a utilized web server and a CPU intensive application. Our experiments can therefore indirectly impact other customer's service only to the same extent as typical use.

Test machines. To test an RFA, we require control of at least two instances running on the same physical machine. As AWS does not provide this capability directly, we used known techniques [25] to achieve sets of co-resident m1.small instances on 12 different physical machines in the EC2 us.east-1c region. Specifically, we launched large numbers of instances of the same type and then used RTT times of network probes to check co-residence. Co-residence was confirmed using a cache-based covert channel. Nine of these were the same architecture: Intel Xeon E5507 with a 4MB LLC. We discarded the other instances to focus on those for which we had a large corpus, which are summarized in Figure 9.

Machine	#	Machine	#	Machine	#
E5507-1	4	E5507-4	3	E5507-7	2
E5507-2	2	E5507-5	2	E5507-8	3
E5507-3	2	E5507-6	2	E5507-9	3

Figure 9: Summary of EC2 machines and number of co-resident m1.small instances running under our accounts.

Each instance ran Ubuntu 11.04 with Linux kernel 2.6.38-11-virtual. For each machine, we choose one of the co-resident instances to play the role of the beneficiary and another one to be the victim. The beneficiary was configured with various benchmarks while the victim had the same Apache installation and configuration as in the local testbed (see Section 5.1). Any remaining co-resident instances were left idle.

We used separate m1.small instances to run the victim load and the RFA traffic generator. We note that despite offering load of 2000 rps on EC2, the achieved load was only around 1500 on average and sometimes slightly less in the presence of RFAs.

Experimental procedure. We chose a subset of the benchmarks (sphinx, mcf, LLCProbe, and bzip2) used in the local testbed for the experiments on EC2. We ran each benchmark on a beneficiary instance while a co-resident victim received requests made by a client load generator as well as an RFA helper, both located on separate EC2 instances that were not co-resident with the beneficiary and victim. We used an intensity of 512 ms and changed the duration of each RFA request to 16 ms, as that was most effective in our experiments. For each benchmark we run the benchmark no RFA, followed by running it with the RFA, and we repeat this three times. (For LLCProbe, each single run of the benchmark was in fact five sequential runs to gather more samples.) This gives 4 data points (10 for LLCProbe). The interleaving of no-RFA and RFA helped limit the effects of unexpected intermittent noise (e.g., from other co-resident VMs outside our control) that may effect measurements. Throughout these experiments the client load generator sends web server requests at a configured rate. We also measure the baseline with no background traffic once at the start of measurements for each benchmark.

Aggregate effectiveness. We start by looking at average performance of the RFA's across all nine machines. Figure 10 depicts the results as normalized average runtimes (average runtime divided by average baseline runtime). Thus higher is better (less slowdown from interference). What we see is that the RFAs provides slight performance improvements across all the instances and, in particular, never hurts average runtime. While the absolute effects are small, they are not insignificant: the RFA improved LLCProbe performance by 6.04%. For the SPEC benchmarks (not shown), we see that the degradation due to the victim (the No-RFA) is, on average, less than observed on the local testbed. This may be due to the different architectures and software configurations, or it may be due to higher contention in the baseline case due to other co-resident instances (owned by other customers). Given the smaller gap between baseline and No-RFA, there is less absolute performance to recover by mounting an RFA. Nevertheless, as a fraction of lost performance, even here the beneficiary receives back a large fraction of its performance lost to interference.

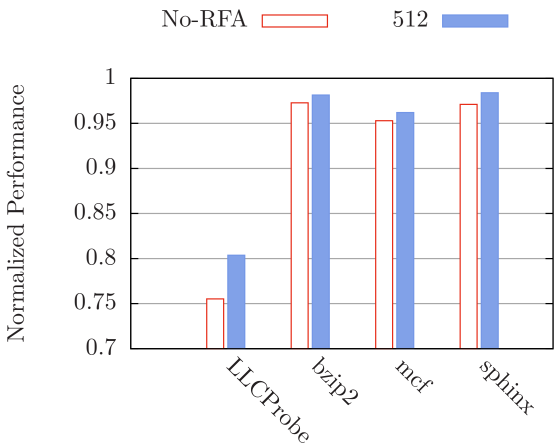


Figure 10: Normalized performance (average baseline runtime over over average runtime) across all machines on EC2 for various workloads.

Per-machine breakdown. To understand the effect further and, in particular, to get a better sense of whether other (uncontrolled) co-resident instances are causing contention, we breakdown the results by individual machine. Figure 11 depicts average runtimes for each machine and for each of the four benchmarks. (The error bars for LLCProbe denote one standard deviation — for the other benchmarks we omitted these due to having three samples.) As it can be seen, the baseline, No-RFA, and RFA performances all vary significantly across the different machines. While we cannot know the precise reason for this, we speculate that it is mostly due to contention from other customer VMs or, possibly, slight differences in configuration and baseline software performance of the distinct machines.

Likewise the benefit of performing an RFA varies by machine. In the case of LLCProbe, RFAs were always beneficial, but the degree to which they improved performance varied. Machine E5507-6 had the highest speedup of 13% from the RFA, which corresponded to decreasing the cost of contention by about 33%. Interestingly, there seems to be little correlation between benchmarks, for example E5507-6 had negative improvement from RFA for the bzip2 and mcf benchmarks. Other machines faired better for SPEC benchmarks, for example E5507-1 had a 3.2% performance improvement under RFAs.

These varied results are not unexpected in the noisy environment of EC2. We draw two general conclusions. First, RFAs can provide significant speedups in the (real-world) environment of EC2, but the benefits will vary depending on a variety of environmental factors. Second, given that the aggregate benefit across all machines is positive, a greedy customer will —on average over the long term— benefit from mounting RFAs.

6. DISCUSSION

Practical dimensions. Deploying a resource-freeing attack like the one explored in the last few sections would be subject to several complicating issues in practice. First, it may be difficult to predictably modify the victim's workload because the victim's normal (pre-RFA) workload may be unknown to the attacker. As shown in Section 5, the amount of extra work required was dependent on the existing workload of the victim. Here, simple adaptive techniques, where workload is continually introduced as long as it improves the beneficiary's performance, may suffice. Moreover, our results suggest an attacker would typically do well to overestimate the RFA intensity required.

Second, it may be that co-resident instances do not have services that are accessible to the RFA helper. As discussed in Section 3 a wide swath of, e.g., EC2 instances run public web servers, and such interrupt-driven workloads are likely to be the most damaging to cache-bound workloads. Even public servers may only be indirectly accessible to the helper, for example if they lie behind a load balancer. Future work might target RFAs that can exploit other avenues of generating a bottleneck resource for the victim, for example the attacker might generate extra contention on a disk drive using asynchronous accesses in order to throttle a victim's I/O bound processes. Such an attack would not require any form of logical access to the victim.

Third, the client workload we experimented with does not reflect all victim workloads seen in practice. For example, if thousands of independent clients submit requests concurrently, the RFA may not be able to effect as much displacement of inbound connection requests (though request processing will still be displaced). Future work might clarify the vulnerability of other victim workloads to RFAs.

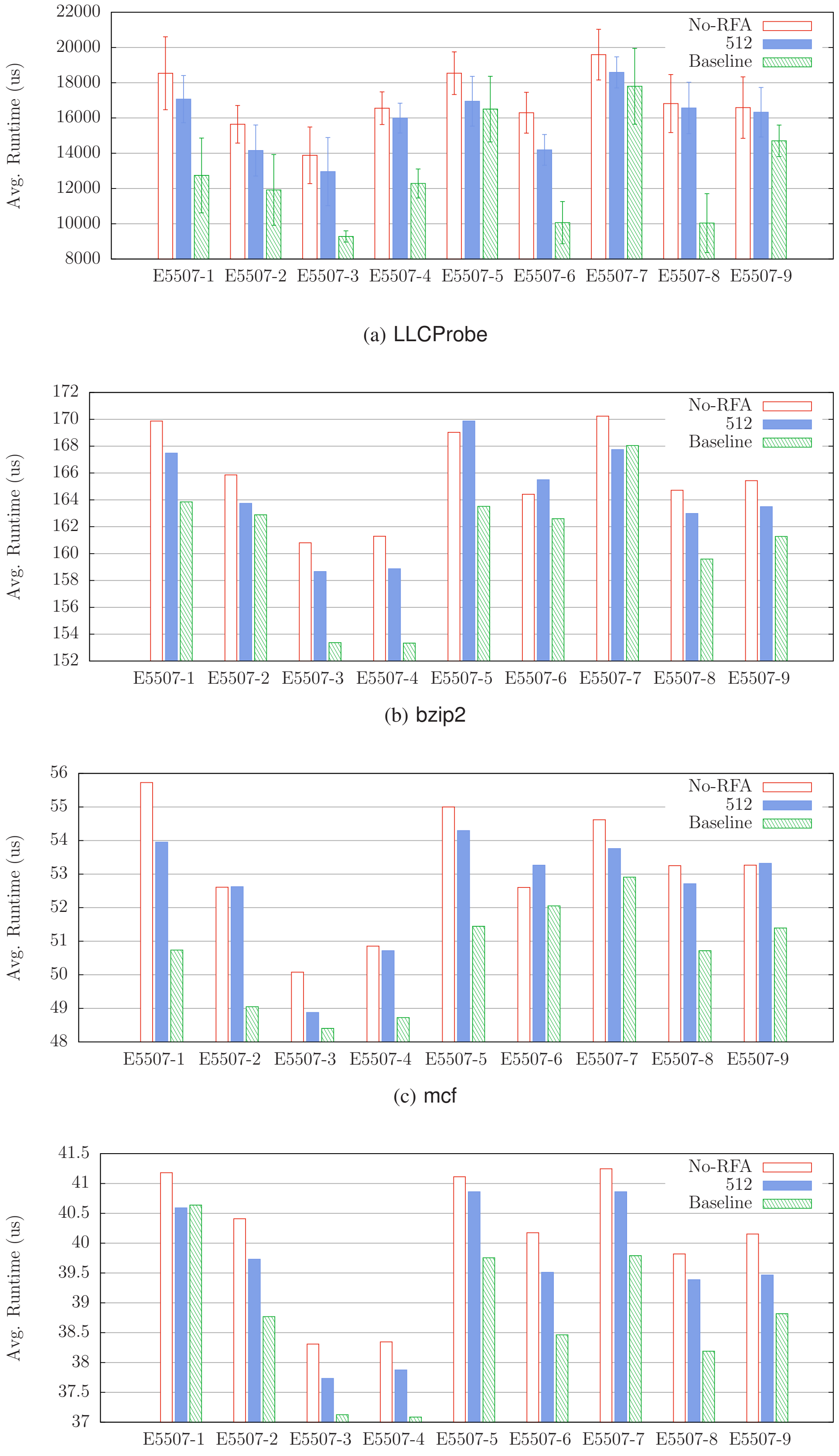
Economics of RFAs. In the setting of public clouds, performance improvement can translate directly to cost improvement since one pays per unit time. For long running jobs, even modest improvements in performance can significantly lower cost. Of course, one must account for the cost of mounting the RFA itself, which could diminish the cost savings. The RFAs we explored used a helper that sends a small number of web requests to the victim. For example, our helper uses only 15 Kbps of network bandwidth with a CPU utilization of 0.7% (of the E5430 as configured in our local testbed). We located this helper on a separate machine. That the helper is so lightweight means that one might implement it in a variety of ways to ameliorate its cost. For example, by running it in places where spare cycles cannot be used for the main computational task or even on a non-cloud system used to help manage cloud tasks. One could also use a cheap VM instance that runs helpers for a large set of beneficiaries, thereby amortizing the cost of the VM instance.

A related issue is that of VM migration. While contemporary IaaS clouds do not enable dynamic migration, customers may move a VM from one system to (hopefully) another by shutting it down and restarting it. The beneficiary could therefore try to migrate away from a contended host instead of mounting an RFA. We view migration and RFAs as two complementary directions along which a greedy customer will attempt to optimize their efficiency. Which strategy, or a combination thereof, works best will depend on the contention, the workload, the likelihood of ending up on an uncontended host, pricing, etc. Understanding the relative economic and performance benefits of migration and RFAs is an interesting question for future work.

Preventing RFAs. To prevent the kinds of RFAs we consider, one could deploy VMs onto dedicated instances. This was suggested in the cloud setting by Ristenpart et al. [25], and subsequently added as a feature in EC2. However, the significant cost of dedicated instances makes it impractical for a variety of settings.

There are two primary methods for preventing RFAs even in the case of multiplexed physical servers: stronger isolation and smarter scheduling. A hypervisor that provides strong isolation for every shared resource can prevent RFAs. This entails using non-work conserving scheduling, so that idleness of a resource allocated to one VM does not benefit another. In addition, it requires hardware support for allocating access to processor resources, such as the cache and memory bandwidth. With current hardware, the only possibility is cache coloring, which sets virtual-to-physical mappings to ensure that guest virtual machines do not share cache sets [14]. This effectively partitions the cache in hardware, which hurts performance for memory-intensive workloads. Finally, it requires that the hypervisor never overcommit and promise more resources to VMs than are physically available, because concurrent use of overcommitted resources cannot be satisfied. While this approach may work, it sacrifices performance and efficiency by leaving resources idle.

A second approach is to apply smarter scheduling. Based on the contention results in Section 4, the hypervisor can monitor the VMs between processes and attempt to schedule those workloads that do not conflict. This approach, often applied to multicore and multithreaded scheduling [6, 10, 29], detects workloads with conflicting resource usages via statistics and processor performance counters, and attempts to schedule them at different times, so they do not concurrently share the contended resource, or on separate cores or packages to reduce contention, as in the case of the LLC.



(a) LLCProbe

(b) bzip2

(c) mcf

(d) sphinx

Figure 11: Average runtimes of LLCProbe, bzip2, mcf, and sphinx benchmarks across all 9 machines. Baseline has no traffic to victim, while No-RFA and 512 RFA have foreground request rate of 2000 rps.

A final idea would be to prevent RFAs by detecting and blocking them. We suspect that this would be very difficult in most settings. RFAs need not abuse vulnerabilities on a system, rather they can simply take advantage of legitimate functionality (e.g., CGI scripts on a web server). Moreover they are stealthy in the sense that it may only require a few requests per second to drive the victim up against a resource bottleneck. A provider or the victim itself would be hard pressed to detect and block RFA requests without preventing legitimate access to the resource.

7. RELATED WORK

Our work builds on past work surveying the performance interference of virtual machines, hardware and software techniques for improving performance isolation, side-channel attacks, and scheduler vulnerabilities.

Performance interference. Numerous works have found severe performance interference in cloud computing platforms [15, 22, 26, 34]. Our study of performance interference focuses more on the worst-case interference in a controlled setting than on the actual interference in cloud platforms. In addition, we measure the interference from pairs of different workloads rather than two instances of the same workload. Finally, our work looks at the impact of multicore scheduling by pinning VMs to a specific core.

Performance isolation. Contention for cache and processor resources is a major cause of performance loss, and many projects have studied resource-aware CPU schedulers that avoid contention [6, 18, 38]. In cache/network contention, these schedulers may place the cache and network workloads on different packages to avoid affecting the cache. Similar work has been done at the cluster level to place jobs [30, 27, 16]. These systems attempt to place workloads that use non-interfering resources together or even to leave a processor idle if interference is bad. These systems would reduce the effect of performance isolation and thus reduce the need for and ability of RFAs to improve performance.

Beyond scheduling, software mechanisms can ensure performance isolation for many hardware resources, including cache [24], disk [11], memory bandwidth [32] and network [28]. Similar to the schedulers described above, these techniques all reduce the performance interference from contention, and if used in a non-work-conserving fashion, can remove the need/benefit of RFAs.

In addition to software techniques, changes to low-level hardware have been proposed to better share memory bandwidth and processor caches [20, 23]. Similar to the software isolation techniques, such mechanisms would reduce the amount of contention and hence the need for RFAs.

Gaming schedulers and allocators. The network/cache RFA works by forcing the scheduler to context switch at much coarser granularities than normal. Similar techniques have been used in the past to game schedulers in Linux [31] and Xen [37] in order to extend the timeslice of a thread. These techniques exploit the difference between the granularity of CPU allocation (cycles) and the granularity of accounting (timer ticks). RFAs are different in that they convert an interactive workload into a CPU-bound workload, and thus affect the priority with which a process is scheduled.

Side-channel attacks. RFAs exploit the lack of isolation to boost performance. Several projects demonstrated side-channel attacks through the shared LLC that can be used to extract information about co-resident virtual machines [25, 36, 35].

8. CONCLUSIONS

Performance isolation proves an elusive goal in cloud computing environments. Despite years of research on how to reduce contention, current cloud providers do not provide strong isolation for reasons of cost and efficiency. We have outlined a new threat that arises at the intersection of imperfect isolation and public clouds: resource-freeing attacks. These are incentivized by the fact that contention can lead to significant efficiency loss, and that translates directly into increased customer costs.

While obviously motivated, we sought to also understand whether they are possible to mount. We therefore performed extensive experiments both on a local Xen testbed and on Amazon EC2. The results show that, for a certain class of benchmarks, a greedy customer can use RFAs to significantly reduce contention for a resource by manipulating a co-resident victim's workload. Having observed gains of up to 13% on live EC2 instances suggests that RFAs are likely to offer improvements for real applications as well.

This is a problem, both for the direct victims of an RFA (that incur increased cost due to spurious requests) and for the cloud provider, which will loose overall efficiency because of the load caused by the extraneous (malicious) gaming of resource allocations. We leave as an open question a detailed exploration of improved resource allocation mechanisms that de-incentivize or completely prevent RFAs.

Acknowledgments

We thank Ari Juels for the initial observations about how cloud resources might be adversarially gamed, which motivated this work, and thank Kevin Bowers and Ari Juels for many helpful discussions about resource-freeing attacks. This work was supported in part by NSF grant 1065134 and by a gift from EMC. Swift has a significant financial interest in Microsoft.

9. REFERENCES

[1] Specjbb2005. http://www.spec.org/jbb2005/.

[2] Graph 500. Graph 500 benchmark 1. http://www.graph500.org/.

[3] Amazon Ltd. Amazon elastic compute cloud (EC2). http://aws.amazon.com/ec2/.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[5] Sean K. Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *MMSys*, 2010.

[6] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *ICS*, 2010.

[7] L. Cherkasova, D.Gupta, and A. Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETERICS Performance Evaluation Review*, 25(2), September 2007.

[8] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Usenix Security*, 2003.

[9] Jake Edge. Denial of service via hash collisions. http://lwn.net/Articles/474912/, January 2012.

[10] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT*, 2007.

[11] Ajay Gulati, Arif Merchant, and Peter J. Varma. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, 2010.

[12] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. In *SIGARCH Computer Architecture News*, 2006.

[14] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM TOCS*, 10(4):338–359, November 1992.

[15] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *IMC*, 2010.

[16] J. Li, M. Qiu, J. Niu, W. Gao, Z. Zong, and X. Qin. Feedback dynamic algorithms for preemptable job scheduling in cloud systems. In *WI-IAT*, 2010.

[17] Linux man page. xentrace(8). `http://linux.die.net/man/8/xentrace`.

[18] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys*, 2010.

[19] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Usenix Security Symposium*, 2007.

[20] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *ISCA*, 2007.

[21] Netcraft Ltd. August 2011 web server survey. `http://news.netcraft.com/archives/2011/08/05/august-2011-web-server-survey-3.html`, August 2011.

[22] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. Understanding performance interference of i/o workload in virtualized cloud environments. In *CLOUD*, 2010.

[23] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.

[24] H. Raj, R. Nathuji, A. Singh, and P. England. Resource management for isolation enhanced cloud services. In *CCSW*, 2009.

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off my cloud: exploring information leakage in third party compute clouds. In *CCS*, 2009.

[26] J. Schad, J. Dittrich, and J. Quiane-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. In *PVLDB*, 2010.

[27] B. Sharma, R. Prabhakar, S. Lim, M. T. Kandemir, and C. R. Das. Mrorchestrator: A fine-grained resource orchestration framework for hadoop mapreduce. Technical Report CSE-12-001, Pennsylvania State University, January 2012.

[28] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[29] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, 2002.

[30] S. Srikantaiah, A. Kansal, and F. Zhao. Energy aware consolidation for cloud computing. In *Proc. HotPowerWorkshop Power Aware Comput. Syst*, 2008.

[31] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Secretly monopolizing the CPU without superuser privileges. In *Usenix Security*, 2007.

[32] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Performance isolation: sharing and isolation in shared-memory multiprocessors. In *ASPLOS*, pages 181–192, 1998.

[33] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI*, 2002.

[34] Guohui Wang and T. S. Eugene Ng. The impact of virtualization on network performance of amazon EC2 data center. In *IEEE INFOCOM*, 2010.

[35] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *CCSW*, pages 29–40, 2011.

[36] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Security and Privacy IEEE Symposium*, 2011.

[37] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. arXiv:1103.0759v1 [cs.DC], March 2011.

[38] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.