CloudShelter: Protecting Virtual Machines' Memory Resource Availability in Clouds

Tianwei Zhang*, Yuan Xu†‡, Yungang Bao†‡, Ruby B. Lee*
*Princeton University
{tianweiz, rblee}@princeton.edu
†State Key Laboratory of Computer Architecture, ICT, CAS
‡University of Chinese Academy of Science
{xuyuan, baoyg}@ict.ac.cn

***Abstract*—We present *CloudShelter*, an architecture to protect virtual machines' memory availability from undesired resource contention on the cloud servers. We introduce a new micro-architectural metric: *Memory Round Trip Time*, to quantify VMs' memory QoS. Using this metric, (1) *CloudShelter* defines new QoS options for customers when launching VMs. These options can guarantee VMs' memory QoS at different levels even when they face intensive contention with co-located VMs; (2) *CloudShelter* periodically monitors VMs' memory QoS at runtime: once QoS violations against customers' demands are detected, *CloudShelter* places this VM into an isolated environment to eliminate contention. *CloudShelter* can reduce 30.1% performance interference from LLC/DRAM contention and 81.6% interference from bus contention[1].**

I. Introduction

Public cloud providers consolidate VMs of different customers to the same cloud servers. Although each VM gets exclusive CPU contexts and memory spaces, they still share the underlying physical resources. There can be significant interference on these resources that degrades the VMs' QoS.

Among these resources, I/O contention has been well studied and solved by mature methods [1], [2]. However, the severity and mitigation of memory contention are relatively less studied. Customers can select memory sizes for their VMs, but they can not control the cache/DRAM capacities and bus bandwidth consumed by their VMs. Contention on these memory resources can degrade the VMs' performance [3], [4]. What is more serious is that a malicious VM can intentionally abuse the memory resources to cause significant performance degradation to the victim VM [5]–[8]. It is challenging to protect VMs' performance from undesired contention of different levels in the memory hierarchy.

We design *CloudShelter*, an architecture which provides memory resource availability protection for customers on demand. The key novelty in *CloudShelter* is the introduction of a new metric, *Memory Round Trip Time* (*MRTT*), designed to quantify the QoS of memory resources consumed by customers' VMs. This metric is based on the observation that the memory system in a cloud server can be treated as a networking system [9]. So we can apply the quantification of QoS in networking to the memory system: the quality of the memory service to a VM is defined as the average latency of its memory accesses. *MRTT* can accurately reflect the VMs' memory QoS and identify the type of resource contention that affects the VMs' performance. *CloudShelter* uses this metric to protect VMs' memory QoS in different aspects.

First, *CloudShelter* provides more comprehensive memory resource options for customers when launching VMs. These options enable customers to specify what levels of memory QoS they desire for their computations. Second, *CloudShelter* provides runtime QoS protection by periodically checking if a protected VM's QoS is compromised due to memory resource contention with co-located VMs. To realize this, *CloudShelter* first measures this VM's runtime *MRTT* when it shares the memory resource with other VMs. Then *CloudShelter* places this VM in a CONTENTION-FREE ZONE (CFZ) and measures its baseline *MRTT*. This CFZ is designed to eliminate co-located VMs' memory interference by partitioning storage resources and prioritizing scheduling resources. By comparing the runtime and baseline *MRTT* values, *CloudShelter* can quantify the effects of resource contention on this VM. If the VM's QoS is breached due to contention on one specific memory resource, *CloudShelter* keeps the protected VM in the CFZ until the resource contention outside the CFZ is relieved.

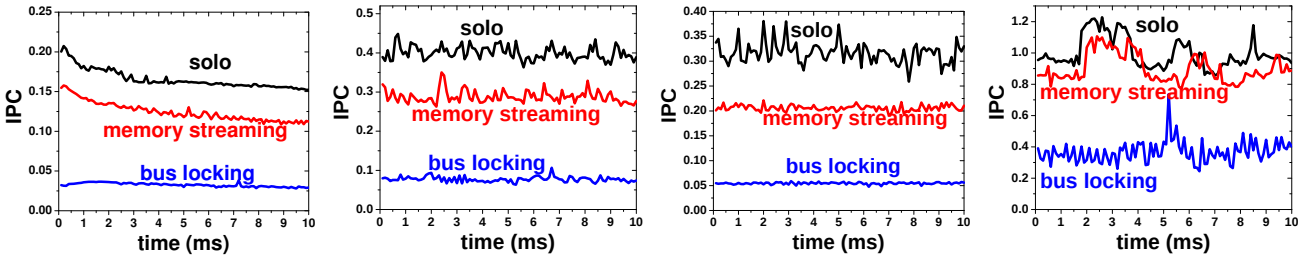
We implement *CloudShelter* in the gem5 simulator [10]. Our evaluation shows that it can reduce 30.1% performance overhead caused by LLC/DRAM contention, and 81.6% performance penalty caused by bus contention. Our simulation results from CloudSim [11] show that *CloudShelter* can also maintain high resource utilization (average 91.9%).

In summary, the contributions of *CloudShelter* are:

- a metric to quantify VMs' memory QoS;
- an architecture which enables customers to specify memory QoS requirements based on their demands, and provides runtime memory QoS monitoring services;
- dynamic resource partitioning and prioritizing mechanisms, to maintain VMs' resource availability;
- QoS-aware protocols for VM scheduling and migration, to achieve high resource availability and utilization.

[1]This work was supported in part by the National Key R&D Program of China under grant No. 2016YFB 1000201, the National Science Foundation under grant NSF CNS-1218817 and the National Science Foundation of China under grant No. 61420106013.

Tianwei Zhang, Yuan Xu, Yungang Bao and Ruby B. Lee. "CloudShelter: Protecting Virtual Machines' Memory Resource Availability in Cloud Computing," IEEE International Conference on Computer Design (ICCD), November 5-8 2017.



(a) mcf (b) omnetpp (c) astar (d) bzip2

Figure 1: Performance degradation due to memory contention.

II. Resource Contention

Memory resource contention among VMs is introduced due to the sharing of the underlying memory system. Basically there are two types of resource contention [8]:

Storage-based contention: This type of contention exists on storage-based resources (*e.g.*, LLC, DRAM bank buffer). When different VMs share the storage-based resources, one VM's data can be evicted from an upper-level component to a lower-level component by other VMs, causing this VM to take longer to fetch the data, thus degrading its performance.

Scheduling-based contention: Scheduling-based resources (*e.g.*, buses, DRAM controllers) arbitrate accesses to storage-based resources. When one VM shares a scheduling-based resource with other VMs, this resource can be temporarily locked or overwhelmed by other VMs. Then this VM's memory request is delayed and its performance is degraded.

Figure 1 shows two cases of memory resource contention. These experiments run on the gem5 in full system mode, with the configurations in Table I in Sec. VI. Specifically we boot 4 VMs on 4 separate cores, sharing the LLC and DRAM. We choose one VM as the target VM, running one of selected SPEC2006 benchmarks. The other three VMs act as contending VMs, running the following programs:

- Memory streaming: this program allocates a large buffer and accesses the data sequentially. It can generate contention on LLC and DRAM (scheduler and bank buffers).
- Bus locking: this program issues atomic accesses to unaligned memory blocks, which generate locks in memory buses and interference other VMs' memory accesses[2].

We measure the target VM's Instructions Per Cycle (IPC) every 0.1ms, for a period of 10ms at stable phases. Figure 1 shows benchmarks' performance when the target VM runs alone (labeled "solo"), runs with contending memory streaming VMs (labeled "memory streaming") and with contending bus locking VMs (labeled "bus locking"). The performance of the target VM are heavily affected by the contending VMs: memory streaming programs can bring 10%-35% degradation to the target VM due to LLC/DRAM contention. Bus locking programs can bring more than 80% degradation to the target VM due to bus contention.

[2]The gem5 simulator does not support bus locking operations, so we implement such feature in gem5 [12]

III. Memory QoS Quantification

To mitigate undesired memory resource contention, it is necessary to quantify the QoS of the memory system in a server. We propose a novel metric to achieve this goal. This metric is inspired by the observation that the hierarchical memory system in a server can be viewed as a networking system [9]: (1) In a memory system, each core sends memory requests to or receives responses from the underlying memory components, while in a networking system, a client sends and receives network packets. (2) The storage-based resources in a memory system act like network servers storing the data. (3) The scheduling-based resources in a memory system act like routers and physical links in a networking system, arbitrating the data.

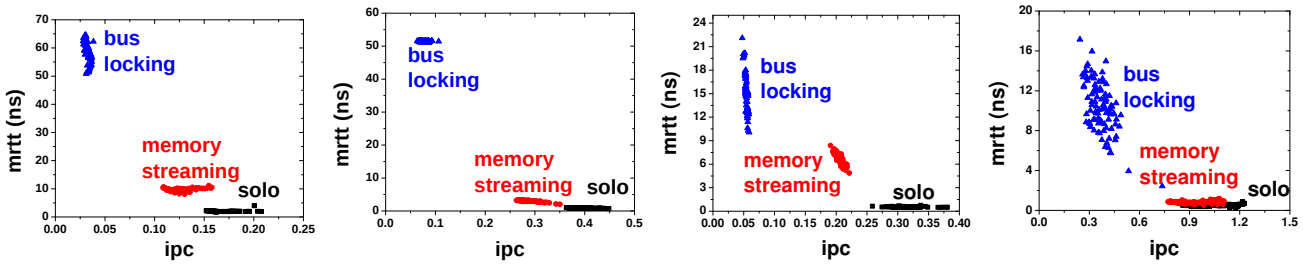
We can apply the concept of networking QoS to the memory system. Network QoS can be quantified by various metrics, *e.g.* error rates, bandwidth, latency. In a memory system, error rate is not a good metric as it is not affected by resource contention. Bandwidth is a performance metric contributed by all the memory layers, so it cannot reveal resource contention on each separate memory layer. We use the memory response latency as the memory QoS metric. It can correctly give one VM's QoS status, and also the severity of contention on each layer of memory resources.

A. Memory QoS Definition

We define *Memory Round Trip Time* (*MRTT*), as *the average response time from the time the CPU core sends a request from a VM, to the time it receives the response.* Figure 4 shows the anatomy of a data memory access transaction. When a core issues a memory request, there are at least three cases, *i.e.*, private cache hit, LLC hit and LLC miss. The round trip time for the three cases are $(t_9' - t_0)$, $(t_9'' - t_0)$ and $(t_9 - t_0)$, respectively. *MRTT* is calculated as the average round trip time for all the memory accesses. We choose *MRTT* to evaluate memory QoS because it has the following advantages:

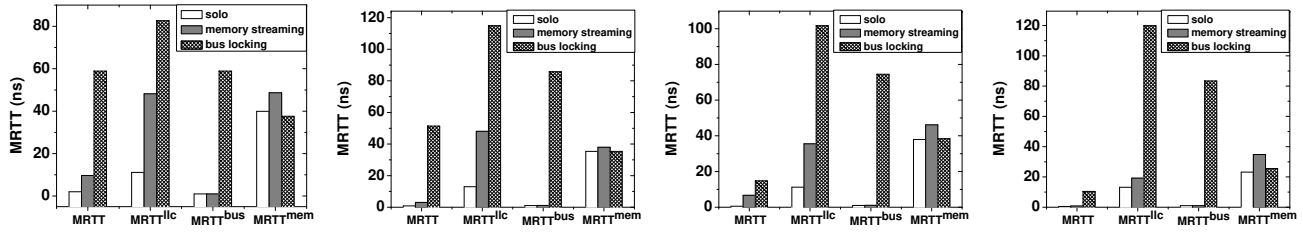
Reflecting overall performance. *MRTT* can be used to quantify a program's performance. One program's average memory latency is negatively correlated with its IPC (Figure 2): if this program has higher latencies, the instructions it completes in a unit time is reduced, so the IPC is decreased.

Identifying performance bottlenecks. In addition to the overall *MRTT*, we can also probe different sub-components



(a) mcf (b) omnetpp (c) astar (d) bzip2

Figure 2: The relationship between the IPC and *MRTT*.



(a) mcf (b) omnetpp (c) astar (d) bzip2

Figure 3: Measurements of each sub-component in *MRTT*

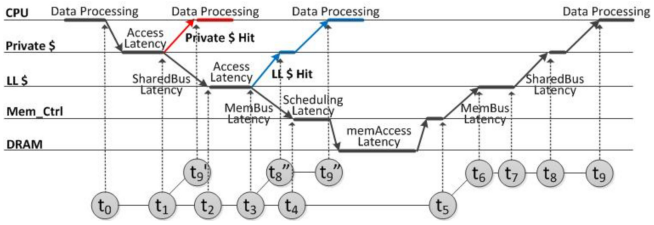


Figure 4: MRTT measurements

of *MRTT* to test if contention happens on some specific resources. We define three *MRTT* sub-component metrics:

- To test if there is LLC contention, we measure $MRTT^{llc}$, defined as the average time from the moment the request heads for the LLC, to the moment the corresponding response returns from the LLC. This time is $(t_8'' - t_1)$ for LLC hit and $(t_8 - t_1)$ for LLC miss in Figure 4.
- To test if there is bus contention, we measure $MRTT^{bus}$, defined as the average time the data travels along the target bus: this time is $(t_2 - t_1 + t_8'' - t_3)$ for LLC hit and $(t_2 - t_1 + t_4 - t_3 + t_6 - t_5 + t_8 - t_7)$ for LLC miss.
- To test if there is DRAM contention, we measure $MRTT^{mem}$, defined as the average time from the moment the request heads for DRAM, to the moment the response returns from DRAM $(t_5 - t_4)$.

Figure 3 shows *MRTT* and each sub-component metric. When the target VM runs with memory streaming programs, $MRTT^{llc}$ increases significantly due to LLC contention (50%–300%). $MRTT^{mem}$ also has a slight increase due to DRAM contention (7%–21%). When the target VM runs with bus locking programs, $MRTT^{bus}$ is increased by 50 to 80 times. Note this interference can also make $MRTT^{llc}$ longer, as $MRTT^{bus}$ is part of $MRTT^{llc}$.

Detecting QoS violation due to memory contention. We can use this metric to check if customers' desired QoS is satisfied. We introduce the *Relative MRTT* ($MRTT_{(relative)}$), defined as the ratio of the VM's *MRTT* when it runs with co-located VMs ($MRTT_{(co)}$), to that when it runs alone ($MRTT_{(solo)}$). A high $MRTT_{(relative)}$ means that the VM's *MRTT* with co-located VMs is much longer than the one when running alone. So resource contention will compromise this VM's QoS. On the other hand, when the $MRTT_{(relative)}$ is close to 1, the resource contention has no effect on the VM's performance, thus this VM's QoS is maintained. $MRTT_{(relative)}$ sub-component metrics are defined similarly.

B. Memory QoS Measurement

Hardware modifications are required to measure memory QoS. To measure the overall *MRTT* online, a new attribute is added in each entry of the Load/Store Unit in each CPU core to record the timestamp the request departs from the CPU. When the corresponding response reaches the CPU, this access latency is calculated as the time interval between the current time and the departure time. In each core, we add a counter (`mrtt`) to record the total *MRTT* of each request, and a counter (`access`) to record the total number of requests, belonging to the VM running on this core. These counters are programmable in the hypervisor to record the values for each VM during any period of time. Thus *MRTT* for this period can be calculated as `mrtt/access`.

Sub-components of *MRTT* can also be calculated similarly. To measure $MRTT^{llc}$ and $MRTT^{mem}$, a new attribute is added to each entry in the Miss Status Handling Register (MSHR) of private cache and Last Level Cache respectively, to record the timestamp of each memory request sent to the LLC or DRAM. Then $MRTT^{llc}$ and $MRTT^{mem}$ are calculated as the intervals between response arrival time and request departure time. To measure $MRTT^{bus}$, a new 8-bit tag is attached to each memory request. This tag is initialized as zero and increases by

one per bus cycle. When the request arrives at the next component through the bus, the scheduling time in the bus can be retrieved from the tag. Our evaluation in Sec. VI shows that an 8-bit tag is long enough to represent the bus latency without overflowing. Each memory component implements several sets of `mrtt` and `access` counters, one set for each core. When there is a context switch, the hypervisor is responsible for resetting these counters.

To measure $MRTT_{(relative)}$, we need to measure *MRTT* of this VM with and without resource contention, respectively. $MRTT_{(solo)}$ ideally should be measured when the target VM is not affected by other co-located VMs. We introduce an isolated environment, CONTENTION-FREE ZONE (CFZ), to eliminate contention on different memory resources. $MRTT_{(solo)}$ can be calculated inside this CFZ, as described next.

IV. CONTENTION-FREE ZONE

A CONTENTION-FREE ZONE (CFZ) is an isolated environment by partitioning storage-based resources and prioritizing scheduling-based resources. It is created for baseline QoS measurement without pausing or interrupting other VMs so their performance will not be significantly affected.

A. Partitioning Storage-based Resources

The basic idea to eliminate storage-based contention is to partition the resource and allocate one part to the target VM exclusively. We show two cases: LLC and DRAM banks.

LLC: LLC is initially shared by all VMs on the same processor socket. When the target VM asks for a CFZ, *CloudShelter* partitions the LLC by ways into two parts: one is allocated to the target VM. The size of this part is determined by the customer's demand. The other part is allocated to the rest of the contending VMs. By doing so there is no interference between the target VM and contending VMs in the socket.

DRAM bank: *CloudShelter* adopts the Virtual Channel Memory (VCM) [13] to eliminate DRAM bank contention. Basically VCM implements a number of fast channel buffers to hold data from different bank buffers in each channel. If the target VM's data is evicted out of the bank buffer by other VMs, the data is still in one channel buffer and can be accessed as fast as accessed from the bank buffer. When the target VM needs a CFZ to protect its required QoS (Sec. V-B), *CloudShelter* partitions these channel buffers into two parts: one is allocated to the target VM. The number of these buffers is determined by the customer's demand. The rest of the buffers are shared by other contending VMs. This can eliminate the interference within the same bank.

B. Prioritizing Scheduling-based Resources

We show how CFZ can eliminate interference on two scheduling resources: the bus and the DRAM controller.

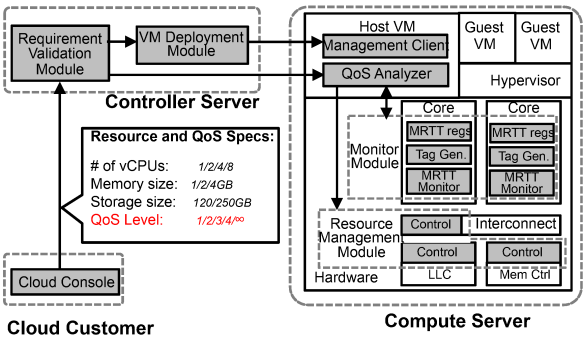


Figure 5: *CloudShelter* architectural overview

Bus: A VM's QoS can be compromised when the bus is frequently locked or overwhelmed by accesses from contending VMs. To solve this contention, we can set three priorities for memory accesses going through the bus: *high*, *middle* and *low*. Initially all the normal memory accesses have *middle* priority and exotic accesses that require bus lock signals have *low* priority. When a CFZ is created, *CloudShelter* gives its memory access *high* priority. *CloudShelter* throttles down the rate of middle-priority normal accesses and low-priority exotic accesses by adjusting their frequency to ensure the bus contention has negligible impact on the target VM.

DRAM controller: The DRAM controller may face contention when it schedules memory requests of different VMs to DRAM banks. *CloudShelter* adopts a priority queuing mechanism that currently supports two priorities: *high* and *low*. Initially, all queues are with the default *low* priority. When a CFZ is required, *CloudShelter* gives its memory requests *high* priority. So these memory requests will win the contention against other VMs in the DRAM controller.

V. CLOUDSHELTER ARCHITECTURE

CloudShelter enables customers to specify QoS requirements for their VMs before launching. At VM runtime, it periodically monitors the VM's memory QoS. Once the QoS is compromised, *CloudShelter* identifies the interference source and reduces the contention to provide required QoS.

A. Architecture Overview

CloudShelter includes three main entities: the customer, the Controller Server, and the Compute Server (Figure 5).

Customer : A customer specifies the configurations to the Controller Server before launching VMs. *CloudShelter* introduces a new launch option: the memory QoS level. The customer can choose one of five memory QoS levels (n = 1, 2 , 3, 4 or ∞), indicating that the VM requires the availability of at least 1, 1/2, 1/3, or 1/4 of the memory resources even under contention with other VMs, or no memory QoS protection at all. Customers can select the memory QoS levels based on how memory intensive their applications are, and how important the availability of memory resources is to the applications' performance. Then *CloudShelter* monitors this VM's $MRTT_{(relative)}$: $MRTT_{(co)}$ is measured when this VM runs with co-located VMs; $MRTT_{(solo)}$ is measured

when this VM runs in an isolated computing environment with the size of memory storage resources equal to $1/n$ (of the LLC capacity and channel buffers) and high scheduling priority (in bus and DRAM controller). *CloudShelter* will check if this $MRTT_{(relative)}$ is smaller than a pre-set threshold (*e.g.*, 150%). If so, this VM's QoS is preserved at Level n. Otherwise, the customer's QoS requirement is not satisfied and further protections will be taken (Sec. V-B).

Controller Server: The Controller Server is responsible for taking requests from customers and allocating VMs to Compute Servers. The `Requirement Validation Module` selects qualified Compute Servers for customers' VMs, which satisfy the VMs' demanded physical resources and the memory QoS level. Then the `VM Deployment Module` is responsible for VM launching and migration.

Compute Server: A Compute Server is a physical machine that hosts the VMs in question. The `Monitor Module` is responsible for collecting and calculating *MRTT* and its *MRTT* sub-component metrics for a given VM. The `Resource Management Module` is the hardware extension which can eliminate the intensive interference on the hardware memory resources. It manages different controls for each shared memory resource in consideration. The `QoS Analyzer` is an agent for querying and interpreting the *MRTT*, checking and identifying contention bottleneck, and triggering the resource controls to eliminate interference.

B. System Operations

VM Launch: When a customer launches new VMs, the Controller Server schedules the VMs to qualified Compute Servers. VM scheduling should achieve both resource and QoS requirements for the customer, as well as high resource utilization for the datacenter.

For resource requirements, we adopt the traditional regular-subscription VM scheduling model [14]: the host server must have more computing resources than the quota required by all VMs hosted on it. For QoS requirements, the host server must be able to create a free CFZ based on the customer's QoS demand, to monitor this VM's QoS. This is a new constraint when allocating VMs.

Algorithm 1 VM_Launch

```
INPUT:
    V^t     /* A set of VMs to be scheduled at time t */
    I       /* A set of Compute Servers in the cloud system */
BEGIN:
    while (V^t ! = ∅) do
        let v* be the VM in V^t that requires the most vCPUs
        let I* be the set of servers capable to host v*
        let i* be the server in I* that has the fewest empty CPUs
        i*.add(v*)
        V^t.remove(v*)
    end while
END
```

To enhance resource utilization, *CloudShelter* consolidates as many VMs as possible on one server, to reduce the number of active servers and energy consumption. We adopt the Best Fit Decreasing approach [15] to solve this approximately (Algorithm 1): when a set of launch requests are fed into the Controller Server, the Controller Server chooses the VM with the maximum vCPU and allocates it to a qualified server with the fewest empty CPUs. It repeats the above procedure until all the VMs are scheduled.

VM Monitoring: The `QoS Analyzer` periodically monitors the VM's $MRTT_{(relative)}$ during VM runtime (Algorithm 2). First, *CloudShelter* checks if the Compute Server is able to create a CFZ based on the customer's demand for this VM. If not, *CloudShelter* uses Algorithm 1 to find a new qualified Compute Server and migrate the target VM to it. *CloudShelter* measures the $MRTT_{(co)}$ when this VM shares memory resource with other VMs. Then it creates a CFZ for this VM (Sec. IV), and measures the $MRTT_{(solo)}$ inside this zone. *CloudShelter* calculates the $MRTT_{(relative)}$ and checks if it is larger than a threshold (*e.g.*, 150%). If so, *CloudShelter* can figure out the resources that incur QoS violations, based on the *MRTT* sub-component metrics. Then *CloudShelter* creates a long-term partial CFZ for this VM, which eliminates contention on these contending resources.

Algorithm 2 VM_Monitoring

```
INPUT:
    v       /* The VM to be monitored */
    i       /* The host server */
BEGIN:
    if (v is in a CFZ) then
        release the CFZ of v
    end if
    if (i is able to create a CFZ for v) then
        measure MRTT_(co) for v
        create a CFZ for v
        measure MRTT_(solo) for v
        release the CFZ of v.
        if (v's QoS is violated) then
                identify the contention resources
                create a partial CFZ for v to eliminate undesired contention
        end if
    else
        i.remove(v)
        call VM_Launch to find a new qualified server i*
        call VM_Monitoring
    end if
END
```

VI. Evaluation

A. Contention Detection and Mitigation

We evaluate the contention mitigation using CFZs.

Experiment settings: We used the gem5 simulator [9], [10] to implement the new hardware of measuring *MRTT* and its sub-component metrics for LLC, bus and DRAM, as well as managing the controls of the memory resources for CFZs. Table I shows the gem5 configurations in our evaluations. Specifically, we simulate a four-core server. It runs a Linux host OS with kernel 2.6.28.4. We use Busybox toolkit [16] to run different virtual machines. Each VM runs an unmodified Gentoo Linux with kernel 2.6.28.4. Each VM occupies a separate core, running SPEC2006 benchmarks, or contention generator programs (*i.e.*, memory streaming or bus locking).

For each experiment, we first use gem5's simpleTiming mode to boot Linux, launch and warmup each workload,

make checkpoints and then switch to Out-of-Order mode. All the results are collected in the Out-of-Order mode.

CPU	4 4-issue Out-of-Order X86 cores, 2GHz
L1I/core	64KB, 2-way, hit = 2 cycles
L1D/core	64KB, 2-way, hit = 2 cycles
Shared LLC	32MB, 16-way, hit = 20 cycles
DRAM	8GB 1 channel, 2 ranks/channel, 8 banks/rank
Disks	4-channel IDE controller, 8 disks

Table I: Simulation Configurations

Results: Figures 6 and 7 show the IPC and *MRTT* of the target VM (running one of the 4 SPEC2006 benchmarks) at different phases when co-locating with contending VMs. Assume this target VM requires a QoS level of 2.

At the first 4 milliseconds, the contending VMs stay idle, so there is no resource contention. *CloudShelter* measures the VM's $MRTT_{(co)}$ outside of CFZs during 0–2ms and places it in a CFZ (1/2 LLC, 1/2 channel buffers, and high priority in bus and DRAM controller) to measure $MRTT_{(solo)}$ during 2–4ms. For some benchmarks, $MRTT_{(relative)}$ is even smaller than 1. This is because in the CFZ the target VM is allocated exclusive, but less resources. So $MRTT_{(solo)}$ is larger than $MRTT_{(co)}$. This $MRTT_{(relative)}$ value indicates that it is safe to place the VM outside the CFZs.

At 4ms, the contending VMs start to execute memory streaming or bus locking programs, generating contention on different levels of memory resources. During 4–6ms, the target VM is outside of CFZs, so its IPC becomes smaller while the $MRTT_{(co)}$ becomes longer. During 6–8ms *CloudShelter* places this VM in a CFZ and measures the $MRTT_{(so)}$. The $MRTT_{(relative)}$ is larger than 150%. So *CloudShelter* needs to protect these benchmarks from memory contention.

After 8ms, *CloudShelter* starts to protect the customer's VM by placing it into a partial CFZ based on the contention source. For memory streaming contention, *CloudShelter* achieves LLC/DRAM bank partitioning and controller priorization in the CFZ; for bus locking contention, *CloudShelter* achieves bus priorization in the CFZ. Then the target VM's performance will not be affected by the contending VMs, and its QoS requirements are guaranteed.

In the real world, *CloudShelter* can measure the $MRTT_{(relative)}$ at a much larger granularity, *e.g.*, every 10s. In each measurement, *CloudShelter* only needs to place the target VM inside the CFZ for 1ms, and then put it back. This can reduce the performance overhead to this VM as well as other co-located VMs. In our experiments, we place the VM inside and outside of the CFZs every 2ms just for evaluations, since the full-system Out-of-Order simulation speed is extremely slow and it is unrealistic to simulate a time period longer than 10s.

B. Performance Overhead

We measure the performance overhead of co-located VMs and the whole server due to CFZ deployment.

Experiment settings: We use the same configurations from Sec. VI-A: 4 VMs run on the same server, and execute the same SPEC2006 benchmark. One VM requests for a CFZ. We measure the average IPC of the other three VMs to test the performance overhead of co-located VMs due to CFZ. We also measure the system performance using the Harmonic mean of Speedups metric (Hspeedup) [17]. A higher Hspeedup indicates better overall system performance.

Results: We consider three cases: the target VM requests for a partial CFZ (1) with 1/2 of storage-based resources; (2) with 1/3 of storage-based resources; (3) with high priority in the scheduling-based resources. Figure 8a shows the average IPC of co-located VMs, normalized to the baseline case without a CFZ. We observe that a CFZ has little performance cost to the host VMs: the worst case is mcf under the CFZ of 1/2 memory resources: the cost is around 5%.

Figure 8b shows the system's Hspeedup normalized to the baseline case without a CFZ. We observe that the existence of CFZs does not bring performance penalty to the system.

C. QoS-Aware VM Scheduling

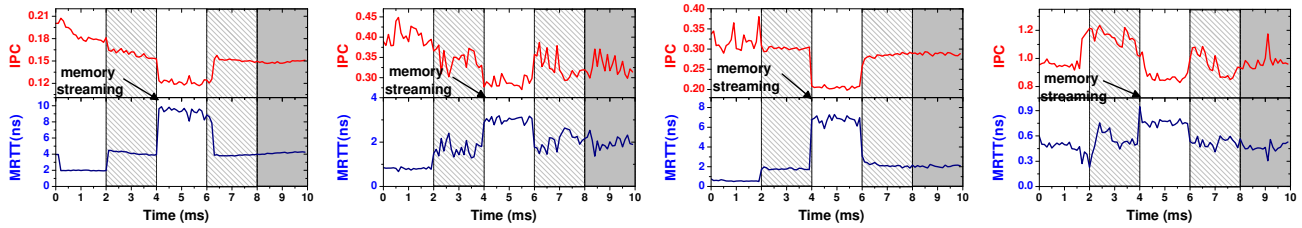
We evaluate the QoS-Aware VM scheduling algorithms proposed in Sec. V-B.

Experiment settings: We use CloudSim [11] to simulate VM scheduling policies. We simulate a cloud system with 100 servers. Each server has 16 physical cores, 64GB DRAM and 5TB disk. We continuously launch VMs until the cloud system is saturated. Each time we launch 10 VMs with randomly selected configurations.

We consider three VM scheduling policies: (1) *Utilization-Aware*: aiming to reduce the number of active servers without considering the QoS requirements; (2) *Random*: allocating VMs to random qualified servers without considering the QoS requirements; (3) *QoS-Aware*: aiming to improve the resource utilization while maintaining VMs' QoS requirement (Sec. V-B). We define the *CPU utilization* as the percentage of physical CPUs occupied by VMs to evaluate the system's resource utilization. We define the *QoS violation* metric as the percentage of VMs requesting but failing to get CFZs.

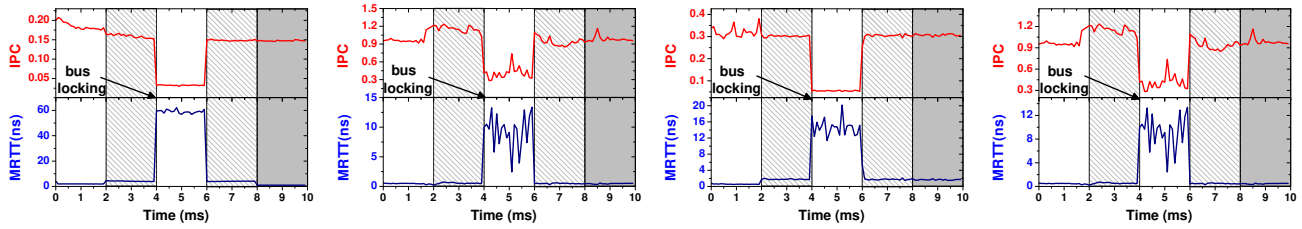
Results: Figure 9a shows the case that 50% of the VMs require QoS protections. For system utilization, *Utilization-Aware* and *QoS-Aware* policies induce constant utilization, and *QoS-Aware* policy has a smaller utilization due to the consideration of QoS requirements. The utilization in *Random* policy is linearly correlated with the number of VMs. For QoS violation, *QoS-Aware* policy has zero violations, while the *Utilization-Aware* policy has very high QoS violations. The QoS violations of the *Random* policy also linearly depend on the number of VMs. We can see that *QoS-Aware* policy can maintain a relatively high resource utilization while satisfying all the QoS requirements.

We adjust the ratio of VMs requesting QoS protections. Figure 9b shows the results when 30% of the VMs require



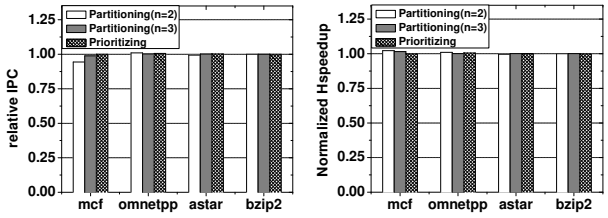
(a) mcf (b) omnetpp (c) astar (d) bzip2

Figure 6: Elimination of memory streaming contention.



(a) mcf (b) omnetpp (c) astar (d) bzip2

Figure 7: Elimination of bus locking contention.



(a) Co-located VMs' performance (b) System's performance

Figure 8: Performance impact on the host system

QoS protections. The utilization in *QoS-Aware* policy is as high as *Utilization-Aware*, with zero QoS violations. Figure 9c shows the results when 70% of the VMs require QoS protections. *Utilization-Aware* policy has a higher utilization than *QoS-Aware* policy. But its QoS violation is also very high. Considering the tradeoff between utilization and QoS violation, *QoS-Aware* policy gives the best results.

VII. Related Work

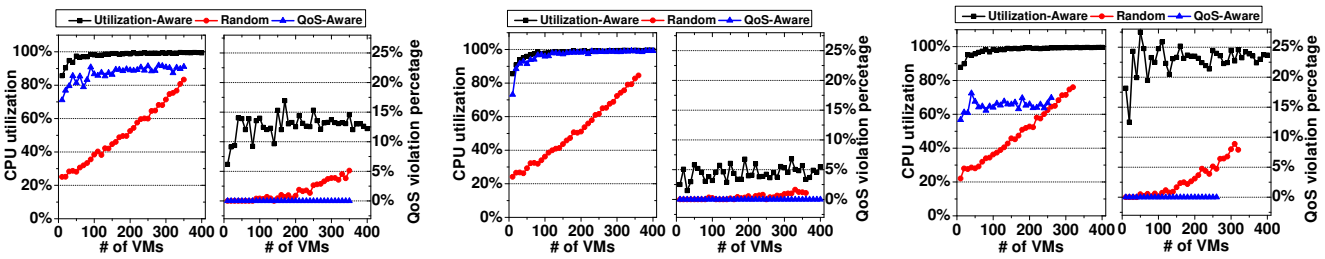
Memory Contention in Cloud Servers. Past work exhibited the memory resource contention in multi-core cloud servers. Tang et al. [3] presented memory resource contention within one domain and between different domains under different domain-to-core mappings. Lo et al. [4] explored resource contention and interference between different latency-critical workloads and batch workloads.

A malicious VM can significantly degrade the performance of other VMs running on the same server by inducing contention on shared memory resources. Grunwald and Ghiasi [5] studied the effect of trace cache contentions on the victim in a Hyper-Threading enabled processor. Woo and Lee [6] proposed memory DoS attacks by frequently flushing L2 caches, saturating or locking internal buses. Moscibroda et al. [7] studied contention attacks on the schedulers of memory controllers. Zhang et al. [8] presented a systematic study of memory contention attacks in virtualized environment, and evaluated the attacks in real cloud. *CloudShelter* can effectively protect VMs' QoS from either benign or malicious resource contention.

Eliminating Contention. One possible solution is to predict the interference between different applications and then schedule them to different servers to reduce contention (*e.g.*, [18]–[20]). Some work proposed methods to monitor the performance of a program or its resource usage, to detect resource contention and schedule these domains wisely [21]–[24]. An alternative is to partition hardware resources(*e.g.*, LLC [25]–[27], DRAM [7], [28]). Different from the above work, *CloudShelter* enables customers to specify the QoS requirements on demand. Then through online monitoring, it is able to figure out *which* resource contention to eliminate and *when* to eliminate the contention, in order to preserve customers' desired QoS.

QoS Metrics and Measurements. Chou et al. [29] proposed a new metric to show applications' performance under a Memory-Level Parallelism scheme. Another popular metric is using Instructions Per Cycles (IPC) [20], [22]. However, IPC can only quantify the overall performance of a VM, so it can be affected by other factors unrelated to memory contention, like interrupts or CPU parallelism. Besides, it cannot reveal which layers of resources are the interference bottlenecks that compromise a VM's QoS. Our proposed *MRTT* can achieve this goal by measuring the *MRTT* sub-component metrics.

To measure a VM's performance without contention, one way is to collect the VMs performance characteristics before it is deployed in the cloud [18], [19]. This cannot provide online interference evaluation. Another way is to measure the target VM's performance online while pausing all other co-located VMs temporarily [20], [30]. This could bring performance penalty to the co-located VMs. Different from



(a) 50% of VMs require QoS protections (b) 30% of VMs require QoS protections (c) 70% of VMs require QoS protections

Figure 9: Resource utilization and QoS violation.

the above work, *CloudShelter* provides online QoS measurements by putting the target VM inside a CONTENTION-FREE ZONE. This can eliminate co-located VMs' interference with reduced impact on their performance.

VIII. CONCLUSIONS

This paper presented *CloudShelter*, a new architectural framework to provide QoS monitoring and protection services to cloud customers. *CloudShelter* enables customers to select different levels of memory QoS based on their memory resource requirements. Then it dynamically monitors VMs' QoS status and resource contention. We proposed a set of approaches to eliminate undesired performance interference on storage-based and scheduling-based resources. As for future directions, we will study the QoS quantification of other resources (*e.g.*, Network, SSD), and the methods to detect and mitigate their contention.

REFERENCES

[1] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman, "Direct device assignment for untrusted fully-virtualized virtual machines," 2008.

[2] Y. Dong, Z. Yu, and G. Rose, "Sr-iov networking in xen: Architecture, design and implementation," in *Conf. on I/O Virtualization*, 2008.

[3] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *ACM Intl. Symp. on Computer Architecture*, 2011.

[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM Intl. Symp. on Computer Architecture*, 2015.

[5] D. Grunwald and S. Ghiasi, "Microarchitectural denial of service: Insuring microarchitectural fairness," in *IEEE Intl. Symp. on Microarchitecture*, 2002.

[6] D. H. Woo and H.-H. S. Lee, "Analyzing performance vulnerability due to resource denial-of-service attack on chip multiprocessors," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[7] T. Moscibroda and O. Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," in *USENIX Security Symp.*, 2007.

[8] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in the cloud," in *ACM on Asia Conf. on Computer and Communications Security*, 2016.

[9] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard)," in *ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2015.

[10] "The gem5 simulator." http://www.gem5.org/Main_Page.

[11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, 2011.

[12] "Intel 64 and ia-32 architectures software developer's manual, volume 3: System programming guide." http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html.

[13] NEC, "64m-bit virtual channel sdram data sheet." http://www.datasheet5.com/download/UPD4565821G5-A80-9JF/236571, 1998.

[14] J. Xu and J. A. B. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *IEEE/ACM Intl. Conf. on Green Computing and Communications & Intl. Conf. on Cyber, Physical and Social Computing*, 2010.

[15] D. S. Johnson, *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[16] "Busybox." https://busybox.net/.

[17] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in smt processors," in *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2001.

[18] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[19] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *IEEE Intl. Symp. on Microarchitecture*, 2014.

[20] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM Intl. Symp. on Computer Architecture*, 2013.

[21] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *ACM European Conf. on Computer Systems*, 2010.

[22] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *ACM European Conf. on Computer Systems*, 2013.

[23] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX Conf. on Annual Technical Conference*, 2013.

[24] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[25] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *ACM/IEEE Intl. Symp. on Microarchitecture*, 2006.

[26] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *IEEE Intl. Symp. on Microarchitecture*, 2006.

[27] "Improving real-time performance by utilizing cache allocation technology." http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[28] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *IEEE Intl. Symp. on Microarchitecture*, 2011.

[29] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *ACM Intl. Symp. on Computer Architecture*, 2004.

[30] A. Gupta, J. Sampson, and M. B. Taylor, "Quality time: A simple online technique for quantifying multicore execution efficiency," in *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2014.