Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds

Mohammad Shahrad*
University of British Columbia

Sameh Elnikety
Microsoft Research

Ricardo Bianchini
Microsoft Research

ABSTRACT

Public cloud providers offer access to hardware resources and users rent resources by choosing among many VM sizes. While users choose the CPU core count and main memory size per VM, they cannot specify last-level cache (LLC) requirements. LLC is typically shared among all cores of a modern CPU causing cache contention and performance interference among co-located VMs. Consequently, a user's only way to avoid this interference is purchasing a full-server VM to prevent co-tenants. Although researchers have studied LLC partitioning and despite its availability in commodity processors, LLC QoS has not been offered to public cloud users today. Existing techniques rely mostly on performance profiling, which is not feasible in public cloud settings with opaque VMs. Moreover, prior work does not address how to deliver differentiated LLC allocations at scale.

In this work, we develop CacheSlicer, the first system that provides cluster-level support for LLC management in a public cloud. We show how to provide LLC allocations in a major public cloud provider to enable differentiated VM categories, from which users select VMs that match their workloads. We integrate it into the Azure VM scheduler and show its effectiveness through extensive evaluations.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;
• **Software and its engineering → Virtual machines**.

KEYWORDS

VM scheduling, resource management, LLC partitioning

*Shahrad was at Microsoft Research during this work.

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3487006

ACM Reference Format:
Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. 2021. Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3472883.3487006

1 INTRODUCTION

Virtual machines (VMs) are the primary resource allocation unit of public clouds. Each VM type typically defines the number of cores, amount of memory, amount of disk space, and disk and network bandwidth that VMs of the type should receive. The cloud user selects the VM type that suits their workload best. Performance-conscious users tend to rent more expensive (aka premium) VMs and expect consistent performance from them. Thus, public cloud platforms carefully manage these resources.

Unfortunately, a key resource is often not managed at all: the last-level cache (LLC). In modern CPUs, cores share the LLC to increase utilization, simplify coherence, and accelerate core-to-core communication [44, 60]. However, the LLC usage is not tightly controlled and a VM may consume a disproportionate amount of space, displacing cache lines from other co-located VMs (aka "noisy neighbor" effect). Thus, in a public cloud, a premium VM may suffer significant LLC interference from co-located VMs, and this interference may change over time [46, 48]. For performance consistency, users currently must rent expensive full-server VMs [82].

Cloud providers could improve performance consistency for smaller VMs by drastically reducing the number of VMs per server. However, this would cause lower utilization and require more servers to be purchased. Alternatively, providers could leverage LLC partitioning schemes proposed in the literature, such as those that use software techniques [31, 61, 66, 78, 80], hardware mechanisms [6, 57, 59, 75], and combinations of software and hardware [11, 62, 70, 72]. However, the prior work has one or more serious drawbacks when it comes to public cloud adoption. A major one is reliance on profiling metrics such as instructions per cycle (IPC) [11, 18, 34, 50], cache miss rate [51, 72, 74], memory bandwidth contention [18], or application-level progress [23, 37, 82]. Such approaches work well when applications and their performance characteristics are known or can be directly inspected. In contrast, public clouds differ from other datacenters in that VMs are almost always opaque to the platform [13]. Cloud

customers are rarely willing to provide application-level performance information about their workloads (via explicit interfaces) or allow the platform to deeply inspect their VMs (via event or system call counters). Consequently from the provider point of view, application-level performance metrics are known only to the customer, and achieving higher or lower IPC says nothing about a VM's performance when it could be just busy-waiting on a lock [22]. Moreover, profiling allows simple adversarial exploits where a VM can get more LLC space (e.g., by artificially increasing its cache misses) [22], choking its co-tenants and relies on dynamic LLC repartitioning, defeating the goal of achieving consistent performance.

Ultimately, there are three research challenges in allocating LLC for performance consistency in the cloud: 1) create workload-agnostic metrics for quantifying LLC contention and reasoning about allocations; 2) define the LLC space allocations of various VM offerings (like CPU and memory allocations, a premium VM type should define the amount of LLC space the VM should receive), so that LLC space usage is maximized without producing excessive contention for the cheaper VM types; and 3) account for LLC allocation as an additional dimension in cluster-wide VM placement in a way that does not excessively harm VM packing.

With these challenges in mind, in this paper we present CacheSlicer, the **first system providing cluster-level support for LLC management of VMs in public clouds**. It introduces new metrics that enable providers to assess contention and manage LLC space without detailed information about VM workloads. CacheSlicer provides differentiated VM categories to users: cache-sensitive (CS) VMs with guaranteed isolation; standard VMs with best-effort isolation; and low-cost VMs with no isolation guarantees. Users can select the category that best matches their workloads' needs.

CacheSlicer comprises three components: 1) a table-based API to specify LLC management policies, 2) an optimization scheme to determine best policies, and 3) an LLC-aware VM scheduler to enforce them. At each server, it uses hardware way partitioning as available from multiple vendors [15, 30, 73]. The table-based API allows convenient control of cluster-level configurations with rich semantics. LLC policy updates are as easy as updating a single configuration file. CacheSlicer's policy optimizer assists the cloud provider in increasing the LLC space for standard VMs without excessively hurting worst-case resource contention among VMs. Finally, the LLC-aware VM scheduler of CacheSlicer maintains a cluster-wide view of current LLC allocations, and places arriving VMs onto servers to make the best use of resources while respecting the VMs' categories.

Our results show that managing LLC space in a cluster-wide manner and optimizing the LLC allocations are highly beneficial, and that CacheSlicer successfully provides the quality of service it promises to all defined VM categories.

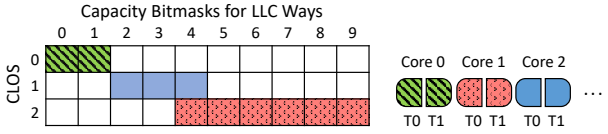


Figure 1: With Intel CAT, each hardware thread is mapped to a class of service (CLOS), and each class is assigned a capacity bitmask (CBM) representing consecutive set of LLC ways.

Our main contributions are the following:

- We propose metrics to assess and manage LLC allocation for opaque VMs in public clouds;
- Using data from the production clusters of Microsoft Azure, we quantify the extent of LLC competition among VMs;
- We propose a scheme that leverages the metrics to optimize the LLC allocations of different VM types/categories;
- We modify the Azure VM scheduler to account for LLC allocation in cluster-wide placement decisions; and
- We evaluate our system extensively using real VM traces from multiple production clusters.

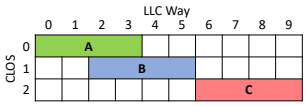
2 BACKGROUND

2.1 LLC Way Partitioning

Hardware way partitioning is supported in several commercial processors from Intel [30], Cavium [72], and Qualcomm [73]. We focus on Intel's Cache Allocation Technology (CAT) because it is widely available on commodity servers in data centers. CAT limits where each hardware thread can allocate its cache lines. Each hardware thread is mapped to a class of service (CLOS), and each class of service is then assigned a capacity bitmask (CBM) specifying the LLC ways where cache lines are allocated. Figure 1 shows a simplified example with only three classes of service (depicted in different colors) and their capacity bitmasks for a 10-way LLC. In contrast for Xeon E5-2600 v3 family [62], which is a commercial processor, LLC has 20 ways and CAT uses 16 classes of service and 20-bit capacity bitmasks. CAT is configured by setting machine-specific registers (MSR) or using libraries [2, 4, 30]. Although CAT has several limitations – including a limited number of classes of service, constrained capacity bitmasks to only consecutive LLC ways, slow update time [21], and reduced cache associativity – it is the most-widely used technique for LLC partitioning.

2.2 VM Scheduling at Microsoft Azure

At Azure, a cluster contains hundreds or thousands of identical servers, and each cluster has a VM scheduler that assigns an arriving VM to one of the servers in the cluster to pack VMs on servers tightly [27]. The VM scheduler solves an online form of multi-dimensional bin-packing [9, 27, 69], and therefore it employs heuristics implemented as a set of scheduling rules applied sequentially.



VM	VM Size (Cores)	Ways	WPC	IWPC	EWPC	RCF
A	1	4	4	2	2.67	0.33
B	2	4	2	1	1.67	0.17
C	1	4	4	4	4	0

Figure 2: These three LLC allocations have the same number of ways, but no two of them are similar.

Each scheduling rule filters or reorders the list of candidate servers, and a rule can be either hard or soft. A hard rule expresses a constraint, and therefore if no candidate server satisfies it, the VM allocation request is rejected resulting in a *scheduling failure*. A soft rule expresses a preference and is ignored if it discards all candidate servers. The scheduler has been in production for several years and proven successful in maintaining low scheduling failures and tight VM packing.

3 MEASURING VM LLC ALLOCATION

In this section, we first define LLC allocation metrics that capture the quality of VM LLC allocation, and then explain the intuition behind them and how they relate to performance. Finally, we quantify the degree of LLC competition in Azure's production clusters using these metrics. In the next section, we show how CacheSlicer uses these metrics to provide different categories of LLC QoS.

3.1 Metrics for VM LLC Allocation

First, to measure the size of LLC allocated to a VM, we count the LLC ways allocated to it since the unit of LLC allocation is a single cache way. To accommodate VMs with different sizes, we normalize LLC allocation using the VM size in CPU cores to define Ways per Core (WPC) for VM_i:

$$WPC_i = \frac{\text{\# Ways allocated to } VM_i}{VM_i \text{ size in cores}} \quad (1)$$

For example, Figure 2 shows three VMs (A, B, and C), each is allocated 4 LLC ways, but their WPC is 4, 2, and 4 respectively since they have different sizes.

Second, VMs A and C have the same WPC, but C has exclusive access to its LLC allocation, which is the requirement for CS VMs. We define Isolated-WPC (IWPC) for VM_i as:

$$IWPC_i = \frac{\text{\#Exclusive ways allocated to } VM_i}{VM_i \text{ size in cores}} \quad (2)$$

Third, to reflect how LLC ways are shared among VMs, we define the Expected-WPC (EWPC) for VM_i as:

$$EWPC_i = \sum_{\omega \in W} \frac{1}{\text{Size in cores of all VMs assigned to } \omega} \quad (3)$$

where W is the set of ways assigned to the VM_i. Intuitively, EWPC measures the number of ways assigned to a VM when proportionally sharing some of those ways with other VMs. For example, in the case of VM A, $EWPC_A = \frac{1}{1}+\frac{1}{1}+\frac{1}{1+2}+\frac{1}{1+2} \approx 2.67$, since the first two ways experience no sharing, whereas the third and fourth ways are shared between VM A (1 core) and VM B (2 cores).

For any VM_i, $IWPC_i \leq EWPC_i \leq WPC_i$. We use these metrics to assess competition on LLC.

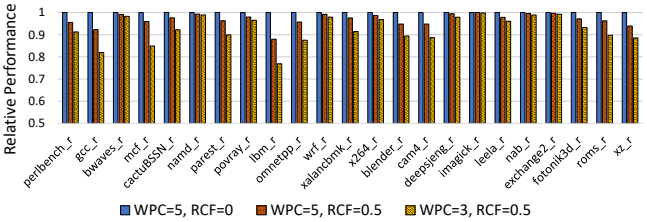


Figure 3: The impact of LLC partition size and sharing using SPEC CPU2017 benchmarks. Each LLC way is 1MB.

Competition among VMs. EWPC is measured in number of ways, which is inconvenient because different VMs may have different numbers of ways. To avoid this problem, we need to compute the normalized amount of competition between a VM and its co-tenant VMs. Thus, we define Resource Competition Factor (RCF) for VM_i as follows:

$$RCF_i = 1 - \frac{EWPC_i}{WPC_i} \quad (4)$$

Here, $0 \leq RCF_i < 1$. Intuitively, RCF quantifies competition as a function of LLC way-sharing. If VM_i does not share ways with any co-tenant VM, then $RCF_i = 0$ and its LLC is isolated with $IWPC_i = EWPC_i = WPC_i$. On the other hand, a VM with RCF close to 1 has high exposure to many competing co-tenant cores that share the LLC with it.

Discussion. So far, we defined the metrics using a snapshot of the hosting server, but these metrics vary over the lifetime of a VM as the set of co-tenant VMs change dynamically with arriving and departing VMs. In the rest of this work, the VM metrics we report are the averages over the VM lifetime. Also, note that there is no loss of generality when we measure VM size in CPU cores. Alternatives include a combination of DRAM size and the number of CPU cores. The motivation for using CPU core count includes the following: (1) CPU core count has a strong correlation with the allocated memory size in cloud VM offerings [16]. (2) A VM with more cores can stress shared LLC more than a VM with fewer cores. (3) The inter-core cache interference is higher than intra-core interference [33].

3.2 WPC, RCF, and Performance

Instead of providing an absolute performance QoS to unknown workloads, we are interested in delivering guaranteed LLC allocation QoS with **monotonic dependence of allocation metrics to performance** [64]. Consider the following statements: (1) Given the same degree of competition, decreasing a VM's LLC partition size cannot improve its performance; and (2) Given the same LLC partition size, increasing competition in a VM's LLC share cannot improve its performance.

To confirm these statements, we run SPEC CPU2017 *rate* benchmarks [10, 38] on the CAT-capable Intel Xeon E5-2620 v4 CPU with 20MB of LLC and 20 LLC ways. Each benchmark uses a core and runs together with the longest

benchmark, *wrf_r*, in three configurations: (1) 5 ways with no overlap ($RCF = 0$), (2) 5 ways with complete overlap ($RCF = \frac{1}{1+1} = 0.5$), and (3) 3 ways with complete overlap ($RCF = 0.5$). Figure 3 illustrates the performance of each benchmark normalized to that of 5 ways with no overlap. We can see that increasing competition (comparing two leftmost bars in each group) or reducing the LLC partition size (comparing the two rightmost bars in each group) never improves the performance of these benchmarks. Without explicitly modeling competition, others have also observed similar trends between sharing and size for SPEC CPU2006 benchmarks [30, 32, 43, 55]. Figure 3 also shows that increasing the competition and reducing LLC size affect different workloads differently. CacheSlicer is designed around the practical constraint that public cloud providers treat VMs as opaque and are unaware of the workloads running on them. Thus, we do not explore this tradeoff and rely on universally monotonic metrics of RCF and WPC. We describe how we use these two metrics in Section 4.1.

3.3 RCF in the Wild

Now that we are equipped with a metric to measure LLC competition, we use it to shed light on the current state of Azure's clusters. Figure 4 shows the cumulative distribution function (CDF) of RCF for all VMs during a day. Each curve corresponds to VMs of a single cluster. All servers in a cluster have the same hardware type, but different clusters have various hardware types[1].

This data is interesting as it shows that server clusters experience a similar LLC competition pattern, despite varying in server type, hosting a diverse mix of VMs, and being located in different regions. The data also shows that the LLC competition is high – roughly 70-80% of VMs in every cluster experience an RCF higher than 80%. As severe LLC sharing is a common issue across the fleet, CacheSlicer's ability to deliver user-controlled LLC guarantees is highly valuable.

4 LLC ALLOCATION POLICY AND OPTIMIZATION

In this section, we first introduce VM categories enabled by CacheSlicer. These categories allow users to choose VM types with the required LLC QoS. To enable this, we first present a compact, flexible, and scalable representation format for LLC allocation policies. Next, we present how we made Azure's VM scheduler LLC-aware. A major piece of CacheSlicer is the policy optimizer which uses the modified scheduler and production simulator with real traces to find best LLC allocations. Finally, we present how we conduct partial sharing of LLC allocations at the server-level.

[1]The total number and types of clusters are business confidential.

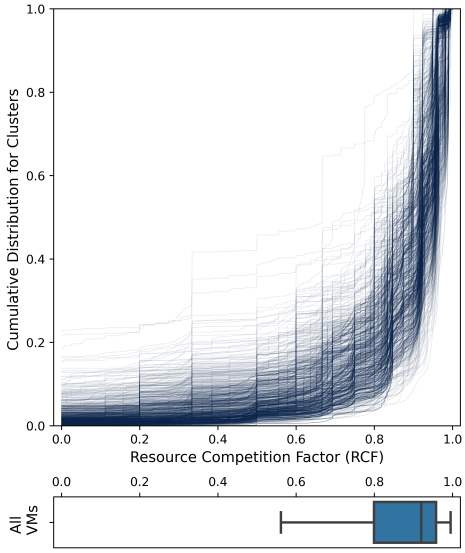


Figure 4: Cumulative distribution of RCF in Azure clusters over a day. Each curve corresponds to VMs in one cluster. The bottom box plot shows the RCF distribution for all VMs.

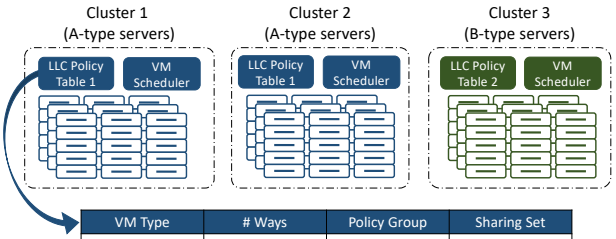
4.1 LLC-Differentiated VM Categories

We define three VM categories and their LLC QoS:

(1) ***Cache-sensitive (CS) VMs***: These VMs benefit from reserved LLC size with full isolation ($RCF = 0, IWPC = EWPC = WPC$) for their lifetime. VMs in this category have various sizes, from which the user can pick. This means that the user can pay less compared to today's full-server VMs required when dealing with CS workloads. At the same time, the provider achieves higher resource utilization by tighter co-location.
(2) ***Standard (ST) VMs***: We define standard VMs to strike a balance between performance and predictability. This is done through optimizing their LLC allocations for larger cache size (WPC) while maintaining low sharing (RCF). We describe this optimization process in Section 4.4.
(3) ***Low-cost (LC) VMs***: Low-cost VMs have no performance predictability guarantees. They share a dynamic portion of the LLC, which we describe in Section 4.4.

While CacheSlicer is built primarily based on these categories, all mechanisms, representations, and APIs are designed to be extensible for an arbitrary number of categories.

Discussion. Note that **cache sensitivity and latency sensitivity are not the same**. While the former captures the significance of LLC for a workload's performance, the latter is an end-to-end QoS expectation from the workload. CS workloads can be latency-sensitive (LS), such as database management systems [47], or not, such as computational fluid dynamics (`437.leslie3d` [29] has highest LLC sensitivity in SPEC CPU2006 [30]). LS workloads can be CS, such as `websearch` [42], or not, such as `memkeyval` which is network bandwidth limited [42]. In fact, Chen et al. [14]



VM Type	# Ways	Policy Group	Sharing Set
T_1	W_1 = *	G_1 = 0	S_1 = {0}
T_2	W_2 = *	G_2 = 0	S_2 = {0}
T_3	W_3 = 2	G_3 = 1	S_3 = { }
T_4	W_4 = 4	G_4 = 2	S_4 = {0, 2}
...	...	...	...

Figure 5: Each cluster has an LLC allocation policy table. The cluster VM scheduler and servers use the same table. Different clusters can have different tables.

found many LS cloud applications "*not highly sensitive to LLC allocations*." Another example is serverless applications that require low execution time but are not CS [63]. CacheSlicer specifically targets delivering hard LLC QoS guarantees to CS workloads.

4.2 LLC Allocation Policies

Early in the project, we realized the need for a declarative way to represent a broad set of LLC allocation policies, including providing a VM with isolated LLC ways and allocating a partition of LLC ways to be shared among specific VMs. This representation expresses the search space the LLC allocation policies explored by CacheSlicer's policy optimizer. Moreover, it must be clean and clear, obviating the need for code modifications when policies change and allowing system developers and administrators to maintain and deploy this capability in production clusters. We present LLC Policy Table (LLC-PT) as a solution to meet these needs. Figure 5 shows a high-level view of LLC-PT containing four fields:

(1) ***VM Type (T)***: Table lookup is based on the VM type. VMs of the same type are managed the same way.
(2) ***Number of Ways (W)***: LLC ways assigned to a VM type. This field either holds a positive integer or is marked unconstrained with an asterisk (*). The latter means that VMs of this type dynamically use unallocated cache ways.
(3) ***Policy Group (G)***: A non-negative integer that allows tying two or more VM types to the same policy. In a valid LLC-PT, two rows having the same G value should only differ in their T fields.
(4) ***Sharing Set (S)***: A list of all policy groups with which VMs of a type can share LLC space. If a VM type's sharing set excludes its own policy group, e.g., for T_3 in Figure 5, VMs of that type cannot have an identical CLOS.

A server uses LLC-PT to map its guest VMs to their corresponding policy groups and sharing sets. We clarify the semantics with the example LLC-PT shown in Figure 5. Assume that the server has ten LLC ways, VM_1 and VM_2 are

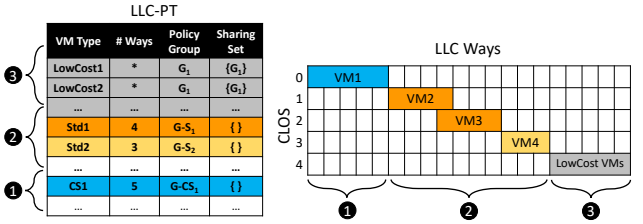


Figure 6: By default, CacheSlicer delivers 3 service categories. Guaranteed full isolation for CS VMs (❶); best-effort isolation for standard VMs (❷); and low-cost VMs sharing the remaining ways (❸).

of types T_1 and T_2 respectively, and both VM_3 and VM_4 are of type T_3. The four VMs are allocated to the same server in their index order. First, VM_1 is scheduled to an empty server and due to being unrestricted (*) can access the entire LLC of its host. Then VM_2 arrives, and since VM_1 and VM_2 have the same policy group (G_1=G_2=0), and sharing within the policy group is allowed ($G_1 = 0 \in S_2$), they both share the entire ten ways. Later, VM_3 arrives and is allocated to two ways (fully isolated). Subsequently, VM_1 and VM_2 can access only the remaining eight ways since their sharing set does not allow sharing with VM_3 ($G_3 = 1 \notin S_1$). Finally, VM_4 arrives and is allocated to two ways (fully isolated) as it cannot share with VM_3 ($G_3 = 1 \notin S_3$) and VM_1 and VM_2 share only six ways.

Figure 6 shows an example LLC-PT implementing CS, standard, and low-cost categories together with a sample corresponding server-level LLC way allocation.

4.3 Making the VM Scheduler LLC-Aware

We modify Azure's VM scheduler to track and enforce LLC allocation policies. We extend the per-server information to include the total number of ways, allocated CS ways, allocated standard ways, and the number of allocated low-cost VMs. Using this information, we add two new scheduling rules (Algorithm 1):

(1) *Full isolation for CS VMs*: This rule ensures that candidate servers for scheduling a CS VM have enough isolated ways. The scheduler rejects the allocation if no such server is available (hard rule). A packing rule follows this rule to pack VMs on servers tightly.
(2) *Best effort packing and isolation for non-CS VMs*: This rule aims to increase packing for standard and low-cost VMs. Among candidate servers that have enough ways to host the new non-CS VM, those with minimum ways left after assignment are kept. The rule also attempts to reduce the sharing between standard VMs, i.e. if there is no server with enough ways to fully isolate a standard VM, it can be overlapped with other standard VMs. A new low-cost VM can be assigned to any server that already hosts a low-cost VM or that has enough LLC ways to create a shared portion for low-cost VMs. This is a soft rule, and the scheduler discards it if the rule returns no server.

Algorithm 1 VM scheduling rules added for CacheSlicer.

```
1:  class Server
2:      ...                                  ▷ LC: Low-Cost, ST: Standard, CS: Cache-Sensitive
3:      MinLCWays ← (1 if HostedLCVMs>0 else 0)
4:      AllocWay ← UsedCSWays + UsedSTWays + MinLCWays
5:      AvailWays ← TotalWays - AllocWay
6:  function GuaranteedLLCForCS(SchedulingRequest, Candidates)
7:      if SchedulingRequest.LLCType != 'CS' then
8:          return Candidates
9:      FilteredCandidates ← {}, ReqWays ← SchedulingRequest.LLCWays
10:     for c ∈ Candidates do
11:         if ReqWays < c.AvailWays then
12:             FilteredCandidates.Add(c)
13:     return FilteredCandidates
14: function BestEffortIsolatedPacking(SchedulingRequest, Candidates)
15:     if SchedulingRequest.LLCType == 'CS' then
16:         return Candidates
17:     FilteredCandidates ← {}, ReqWays ← SchedulingRequest.LLCWays
18:     if SchedulingRequest.LLCType == 'LC' then
19:         for c ∈ Candidates do
20:             if (ReqWays < c.AvailWays) or (HostedLCVMs>0) then
21:                 FilteredCandidates.Add(c)
22:     else if SchedulingRequest.LLCType == 'ST' then
23:         MinScore ← ∞
24:         for c ∈ Candidates do
25:             Score ← ReqWays - c.AvailWays + c.MinLCWays
26:             if Score < MinScore then
27:                 MinScore←Score, FilteredCandidates ← {c}
28:             else if Score==MinScore then
29:                 FilteredCandidates.Add(c)
30:     if isempty(FilteredCandidates) then
31:         return Candidates                ▷ Making it a soft rule
32:     return FilteredCandidates
```

The candidate servers filtered by these rules are passed to the remaining rules of the VM scheduler to determine the final candidate. If there were more than one final candidate, one of them is randomly selected.

4.4 Optimizing LLC Allocation Policy

The optimization targets LLC allocations to standard VMs, because CS VMs are allocated fully-isolated shares (hard constraint), and Low-cost VMs dynamically share remaining LLC space subject to a minimum LLC allocation requirement. In contrast, determining a good allocation for ST VMs is challenging. Standard VMs are allowed to have partial LLC sharing with each other. They could be allocated very small shares (low WPC) with low sharing ($RCF \rightarrow 0$). On the flip side, they could be assigned very large shares (high WPC) with high sharing ($RCF \rightarrow 1$) as well as excess pressure on low-cost VMs. To strike a balance between these competing factors, we optimize the LLC allocation policy using an offline algorithm that exploits history of cluster-wide VM arrivals/departures and distribution of VM sizes and categories. As the mix of VMs changes over time, the optimizer should be re-run on fresh distributions to ensure the most optimal allocations. Figure 7 shows an overview of the optimizer.

Algorithm 2 describes how the allocation optimizer iteratively improves LLC allocations, and Table 1 defines the

Algorithm 2 Allocation optimization algorithm.

```
1:                  ▷ N: parallelism, I_max: maximum optimization iterations
2:                  ▷ λ: annealing factor (temperature), C_Target: target cost
3:  function AllocationOptimizer(N, I_max, λ, C_Target)
4:      C_Min ← ∞                                   ▷ global minimum cost
5:      P_0 ← {p_0^0}                ▷ starting with standard fair share allocation
6:      for i ∈ [0, I_max] do
7:          L_i ← RunSimulatorInParallel(p ∈ P_i)
8:          for l ∈ L_i do
9:              C_i.append(CalculateCost(l))
10:         [C_min, j] ← min(C_i)                   ▷ min cost with its index
11:         if C_min < C_Min then
12:             [p*, C_Min] ← [p_i^j, C_min], n_i ← p*
13:         else if random([0,1]) < λ^i then
14:             n_i ← p_i^j
15:         else
16:             n_i ← p*
17:         if C_Min ≤ C_Target then
18:             break
19:         P_{i+1} ← AllocationRecommender(n_i, N)
20:     return [p*, C_Min]
```

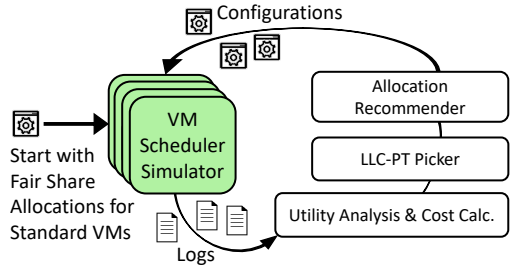


Figure 7: Allocation optimizer overview.

algorithm's variables. The optimization starts with a *linear fair share* assignment for standard VMs, where they are assigned ways proportionally to their number of cores:

$$W_i = Round(\text{ways}_{\text{server}} \times \frac{\text{cores}_i}{\text{cores}_{server}}),\ \forall T_i \in \text{Standards} \quad (5)$$

The *incremental allocation recommender* (line 19 and also Figure 7) incrementally increases/decreases the numbers of ways W_i to form new policy tables. It also enforces the following constraints increasing/decreasing ways: (1) Each VM must receive a positive number of LLC ways; (2) The allocated ways to a VM type cannot exceed ways available on each server; and (3) Within the same VM family, larger VM types cannot have fewer ways. For instance, in a hypothetic VM family of A, assigning three ways to the single-core A_0 type while allocating two ways to the two-core A_1 is invalid. The optimizer employs the Azure VM scheduler simulator to evaluate candidate tables and uses the results to calculate utility metrics, such as WPC, IWPC, and RCF for all VMs. Note that using the VM scheduler simulator is necessary to consider all other resources allocated to VMs. It then combines metrics in a cost function to assess the allocation fitness, as we discuss below. It uses the simulated annealing technique [36] to avoid local minima – at each iteration, if the global minimum cost was improved by an allocation, it

Symbol	Definition
P_i	set of policy tables simulated in optimization round i
L_i	simulation logs for P_i
n_i	proposed base allocation for round i+1 of optimization
N	degree of parallelism for allocation optimizer
$p^\star$	optimized policy table

Table 1: List of symbols.

becomes the new optimum allocation; if not, with a decreasing probability, the best allocation of that round (as opposed to best global allocation) becomes the base for the next iteration. To accelerate the convergence, the optimizer explores multiple (N) candidate policy tables in parallel. In this paper, we use $N = 8$ but other values also work well.

The optimizer's flexibility stems from incorporating several objectives. For example:

$$C = \overbrace{\frac{1}{\text{Avg. ST WPC}}}^{\text{standard size}\uparrow} + \overbrace{(\text{Avg. ST RCF})}^{\text{standard sharing}\downarrow} + \overbrace{(\frac{\text{LC with WPC} < 1}{\text{LC Count}})}^{\text{low-cost pressure}\downarrow} \quad (6)$$

This cost function favors an allocation that gives more ways to standard VMs in such a way that reduces the LLC competition among standard types and limits the impact on low-cost sizes. In this work, we use a slightly more advanced cost function:

$$C = \overbrace{\frac{1}{\text{Avg. ST WPC}}}^{\text{standard size}\uparrow} + \overbrace{(\text{Avg. ST RCF})}^{\text{standard sharing}\downarrow} + \overbrace{(\frac{\text{LC with WPC} < 1}{\text{LC Count}})}^{\text{low-cost pressure}\downarrow} + \underbrace{2 \times (\frac{\text{ST with RCF} > 0.2}{\text{ST Count}}) + 10 \times (\frac{\text{ST with RCF} > 0.8}{\text{ST Count}})}_{\text{excess standard sharing}\downarrow\downarrow} \quad (7)$$

It has two additional shaping terms to control the maximum RCF of standard VMs; the weights 2 and 10 reflect the importance we place on standard RCFs being lower than 20% and 80%, respectively. Figure 4 shows that RCF values above 80% are currently very common. We want to restrain the optimizer from delivering standard VMs the RCF they are already experiencing. A cloud provider can use other cost functions based on its revenue and user satisfaction models.

4.5 Partial LLC Sharing at the Server Level

Over-subscription of LLC ways is expected with arbitrary-sized CS and upsized standard VMs. As stated earlier, this means that LLC allocations of standard VMs could partially overlap. The LLC-aware scheduler mitigates unnecessary sharing and the optimizer assesses and limits sharing. But if sharing occurs, there needs to be a server-level placement policy to manage potential overlaps.

Finding the best server-level overlapped allocation is a complex problem involving optimizations and heuristics [25]. CacheSlicer uses a simple heuristic algorithm which we call

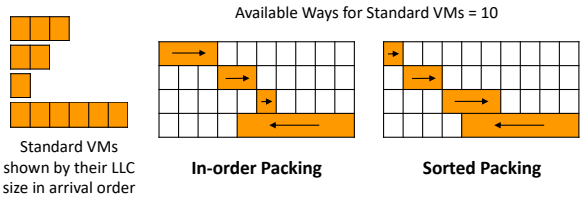


Figure 8: Standard VM allocations might need to be overlapped. Here, 4 standard VMs requiring 12 ways are fit into 10 ways using two simple algorithms.

sorted packing. The idea is to minimize the chance of overlapping between small LLC allocations and we show its effectiveness in Section 5.5.3. It works by sorting standard VMs in ascending order of their number of ways and placing them back to back inside the standard-only partition. The size of this partition is determined by the number of allocated ways to CS VMs and the minimum LLC ways required to serve low-cost VMs. The placement direction toggles if there are not enough ways remaining. Also, each standard VM's maximum way count is limited to the standard partition size. Figure 8 shows an example of this algorithm as well as the case where VMs are not sorted, and instead packed in their arrival order (*in-order packing*). The standard partition has ten ways here. Note that defragmentation of LLC ways after a VM termination is as straightforward as few MSR updates.

4.6 Supporting Multiple CPU Sockets

When a VM spans over multiple sockets, it can experience significant performance degradation [8, 83]. This is why modern cluster schedulers make sure that VMs not requiring more cores than a single CPU are assigned and pinned to just one socket [5, 41, 79]. For a large VM that has to span over sockets, the remaining cores are allocated to another socket. In such a case, we treat the LLC allocation to those remaining cores the same as a small VM assigned to a socket.

5 EVALUATION

CacheSlicer manages the LLC at the cluster level to enable delivering differentiated LLC QoS categories to VMs. We perform realistic large simulations to extensively evaluate design trade-offs at that scale and stress the system with various deployment scenarios. Towards this goal: (1) we simulate production-scale clusters (720 servers); (2) we use real VM distributions from five CAT-capable production clusters from three geographical regions; (3) we use the hardware configurations of those clusters, specifically, each server we study has 40 cores and 40 LLC ways; and (4) we conduct the simulations using Azure's VM cluster scheduler simulator. This simulator is faithful to the real scheduler and is used by Azure to evaluate any changes to the scheduler. It has also been used in previously published studies [7, 16]. A recent study [27] discusses the simulator's fidelity.

LLC Requirements	Cores	1	2	4	8	16	32
Fair-Share ($WPC = 1$)		1	2	4	8	16	32
Small-Fast	LLC Ways	2	4	8	16	32	40
Large-Slow		4	8	12	16	20	24

Table 2: Isolated LLC ways for CS VMs.

For the standard VMs, we tag VMs with four cores or more as standard, since larger VMs are more expensive and tend to demand higher QoS for example to host high-end applications that can benefit from larger LLC allocation. We analyze the impact of changing the set of standard VMs in Section 5.5.1. For the CS VM category, we tag two of Azure's VM families as CS, each including six VM sizes. These twelve CS VM types constitute 3.56% of the total VM count and 4.16% of the total allocated cores in the five clusters. For the LLC requirements, we study the three settings listed in Table 2. The number of isolated LLC ways is linearly proportional to the number of cores in the *Fair-Share* setting, but starts and grows differently for the *Large-Slow* and *Small-Fast* settings. Averaging across all CS VMs, a CS VM has 3.75 cores but has 3.75, 8.35, and 7.3 LLC ways for Fair-Share, Large-Slow, and Small-Fast, respectively. CS VMs under the two latter settings receive larger isolated LLC allocations and therefore reduce the number of LLC ways available for standard VMs. We study the impact of the LLC requirements and the percentage of CS VMs in Section 5.5.2. By default, we assume sorted packing for the server-level partial LLC sharing and compare it against in-order packing and brute force in Section 5.5.3. Finally, our study does not include potential VM lifetime variations as a result of LLC allocations. Since VM behavior is opaque to the provider, a real deployment is needed to know if the VM lifetimes and mix will be impacted by LLC-aware allocations.

5.1 Understanding Allocation Optimization

We evaluate the internals of the policy optimizer assuming no CS VMs for clarity since their LLC requirements are fixed constraints and we study their impact in subsequent sections. Figure 9 shows the evolution of the main cost function components (Equation 7) during the optimization. Each mark represents the outcome of simulating two weeks of deployment with a different candidate policy. The optimization path is depicted with the dashed line starting from the lower-left corner of the plot with the fair share LLC assignment for standard VMs where the number of ways is proportional to core counts (described earlier in Section 4.4). As the optimizer incrementally explores candidate LLC allocation policies, those improving the cost function the most become the basis for further explorations. This translates into improving the WPC of standard VMs while maintaining a low sharing (RCF) for them and also low pressure on low-cost VMs.

As seen in Figure 9, most of the optimization benefits come from early iterations, and the optimization path eventually converges. Decremental allocation explorations can also be seen in the figure. For instance, the fall right after the tenth

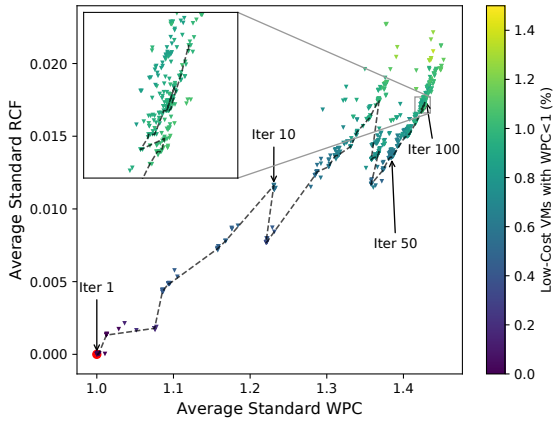


Figure 9: Evolution of cost function components during ST VM LLC optimization. Each dot corresponds to an explored policy. The optimization path is depicted with a line.

iteration shows how a backward step improved the solution by slightly lowering the WPC but considerably reducing the RCF. Performing such backward steps is necessary to compensate for prior suboptimal policy changes. An empirical 10% probability to take such backward steps helped us strike a balance between policy exploration quality and convergence speed.

By default, the allocation optimizer treats each standard VM type separately and modifies its way allocation independently. Another approach is to assign the same number of ways to standard VMs of the same size. This joint optimization of same-size standard VMs prevents over-fitting to a specific VM type distribution and prevents unintentional incentives for users to migrate to different VM types of the same size. In addition, joint optimization reduces the search space, leading to faster convergence. On the other hand, optimizing standard VM types independently allows the cloud provider to take full advantage of the type popularity information and possibly serve the popular VM types better.

5.2 Differentiated LLC Allocation

We show how CacheSlicer delivers different LLC allocations to CS, standard, and low-cost VMs. To do so, we perform the policy optimization on a 720-server cluster with 3.6% of VMs belonging to the Small-Fast CS category. Figure 10 shows the policy table for all VM types after the optimization. We use this policy and simulate the cluster for two weeks. The warm-up process to reach the desired level of VM packing and core allocation (80% of cores allocated to VMs) takes less than a day; thus, the cluster runs at steady state for most of the two-week period mimicking production clusters.

Figure 11 shows the distribution of IWPC and EWPC for all VMs across the cluster, with and without CacheSlicer. Without CacheSlicer, all LLC ways of each server are shared among its VMs. Since each server has 40 cores and 40 LLC ways, a VM has EWPC of at least one when running on a fully

VM Type	# Ways	Policy Group	Sharing Set
LC_1C	*	LCG_1	$\{G_1\}$
LC_2C	*	LCG_1	$\{G_1\}$
...	...	...	...
S_4C	6	SG_1	{ }
S_8C	10	SG_2	{ }
S_16C	16	SG_3	{ }
S_32C	33	SG_4	{ }
S_40C	40	SG_5	{ }

VM Type	# Ways	Policy Group	Sharing Set
CS_1C	2	CSG_1	{ }
CS_2C	4	CSG_2	{ }
CS_4C	8	CSG_3	{ }
CS_8C	16	CSG_4	{ }
CS_16C	32	CSG_5	{ }
CS_32C	40	CSG_6	{ }

Figure 10: The optimized LLC-PT used in Section 5.2.

packed server, and greater than one when the server is not fully packed. When sharing the entire LLC, a VM experiences an IWPC of zero even when it has a single co-tenant.

In contrast, CacheSlicer provides three LLC QoS classes. All CS VMs enjoy guaranteed full isolation of their required LLC space ($WPC = EWPC = IWPC$). Standard VMs constitute 20.2% of VMs and receive 61.1% of allocated cores since VMs with four or more cores are assumed standard VMs. Moreover, CacheSlicer upsizes standard VMs ($WPC > 1$) using the optimized LLC allocation policy, and they receive best-effort isolation as reported in Figure 11. Low-cost VMs still experience low isolation since they share the same LLC space, and their EWPC decreases due to potential CS or standard co-tenant VMs.

5.3 LLC-Aware VM Cluster Scheduler

CacheSlicer's LLC-aware VM scheduler, described in Section 4.3, delivers full isolation to CS VMs without any allocation failures. However, the primary role of the LLC-aware scheduler is in the process of optimizing way allocations. Figure 12 compares the EWPC and IWPC of standard VMs using LLC-agnostic and LLC-aware schedulers. Here, the LLC-agnostic scheduler denotes the current scheduler, and we use the same configurations and VM arrival sequence to test it. We simulate the scenario from Section 5.2 with Small-Fast as well as the Large-Slow CS way allocations.

The LLC-aware scheduler improves EWPC and IWPC 2.7% and 2.5% for the Small-Fast setting, and 14.1% and 12.6% for the Large-Slow setting, respectively. These benefits are due to our soft rule (Section 5.2), which reduces unnecessary LLC contention, thereby providing standard VMs with larger LLC space while maintaining good isolation among the upsized standard VMs and low impact on low-cost VMs.

5.4 Behavior Differences Across Clusters

CacheSlicer's optimizer uses a VM type distribution during simulation of candidate policy tables. As stated earlier, we base the optimizations on the combined VM type distribution of five production clusters from three geographical regions. We refer to the LLC-PT resulting from this optimization approach as "globally optimized". Here, we answer two questions: (1) how do the results differ across clusters when we schedule their VM arrivals with the globally optimized LLC-PT? (2) how much extra gain would each cluster attain when using a "regionally optimized" table, i.e. an LLC-PT that was optimized based on its region's VM type distribution? We do not consider any finer granularity than a region in producing an LLC-PT; as a customer's VMs can be mapped to any of the clusters in a region, having different policy tables for them can cause allocation inconsistencies beyond user's control, defeating the purpose of CacheSlicer.

Figure 13 shows the average EWPC and IWPC for standard VMs with globally optimized (left/blue bars for each cluster) or regionally optimized (right/green bars) LLC-PTs. Each cluster's region is listed inside parentheses. These results exclude CS VMs for simplicity. It is interesting to see that the average EWPC and IWPC are almost exactly the same for these clusters. The reason is that RCFs are close to 0 for them. To answer the first question we pose above, we compare the blue bars to each other and the green bars to each other. We can see differences in both comparisons, but especially across the green bars, because regional optimization is more closely related to the VM type distributions that clusters actually experience. To answer the second question, we compare each blue bar to its green counterpart. The results show that regional optimization always leads to at least as much LLC space and isolation as global optimization for standard VMs.

Deciding between global and regional optimization depends on the provider's desire to maintain inter-region allocation consistency. To prevent over-fitting, we conservatively based our analyses on global tables. But, as shown above, regional optimization leads to better results.

5.5 Sensitivity Studies

As we propose new VM categories, their usage pattern is unknown. In this section, we conduct sensitivity studies to evaluate CacheSlicer in various scenarios to assess the impact of the standard VM classification, the CS VM way allocations, and the server-level LLC sharing algorithm.

5.5.1 Sensitivity to Standard VM Classification. Earlier in Section 5, we selected non-CS VM types having four or more cores as standard VMs. For the combined distribution of the five clusters, this choice translated to 21.1% of the VMs and 63.9% of the allocated cores being standards. Each standard VM had an average of 8.04 cores.

To assess the impact of the choice of standard VMs, we now study a different mix of VM types with a similar percentage of standard VMs (21.6%). However, this new mix represents 22.5% of the allocated cores, where each standard VM has an average of only 2.76 cores. We refer to these two definitions for standard VMs as *large* and *small*, respectively.

The policy optimization increases the WPC of small standards on average to 2.18 (EWPC: 2.14, IWPC: 2.11), whereas for large standards, the optimized allocations had an average WPC of just 1.43 (EWPC: 1.40, IWPC: 1.38) as shown in Figure 9. This is expected as smaller VMs experience a greater WPC increase when given the same number of ways.

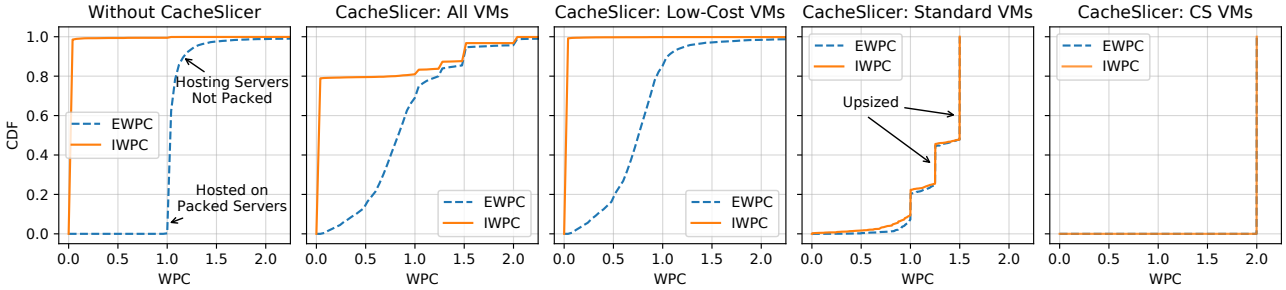


Figure 11: The distribution of VMs' expected and isolated LLC space with and without CacheSlicer. The allocated LLC space is fully isolated for CS VMs, highly isolated for standard VMs, and shared for low-cost VMs.

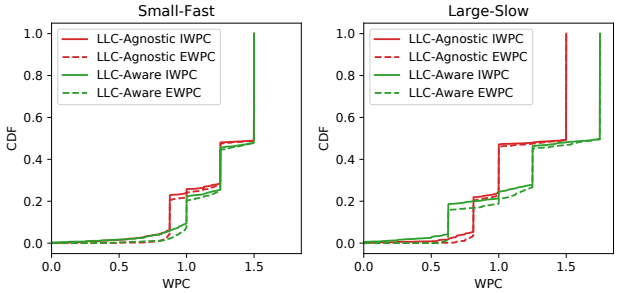


Figure 12: LLC-aware scheduler improves EWPC and IWPC of ST VMs by 2.7% and 2.5% under the Small-Fast setting, and by 14.1% and 12.6% under the Large-Slow setting, respectively.

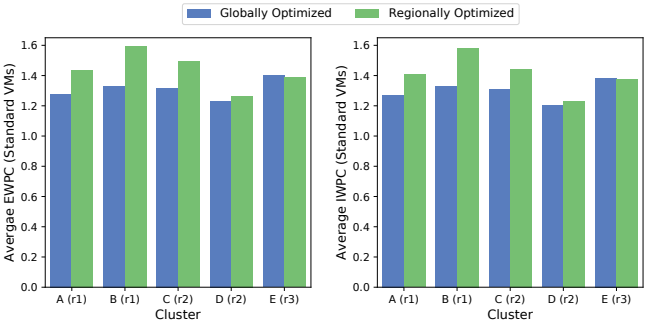


Figure 13: Using cluster-specific (regional) VM distributions instead of a global distribution improves LLC allocations.

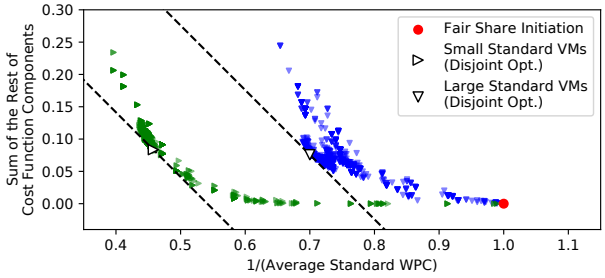


Figure 14: Decomposing the optimizer's cost function. Independent of the choice of standards, the Pareto-optimal allocation resides where the drawbacks of upsizing equal its benefits.

CS Setting		Fair-Share	Small-Fast	Large-Slow
Avg. WPC for Cache-Sensitive VMs		1.0000	1.9920	3.4523
Cost function value	Unoptimized	1.0000	1.0165	1.0171
	Optimized	0.8232	0.8506	0.8577
Avg. WPC for Standard VMs	Unoptimized	1.0000	0.9996	1.0000
	Optimized	1.3340	1.3332	1.3993
Avg. RCF for Standard VMs	Unoptimized	0.0000	0.0033	0.0050
	Optimized	0.0175	0.0238	0.0263
Avg. Isolation (%) for Standard VMs	Unoptimized	100.00	99.408	99.138
	Optimized	96.867	95.762	94.907
Avg. WPC for Low-Cost VMs	Unoptimized	15.197	13.757	13.245
	Optimized	11.663	10.104	9.9217
Avg. RCF for Low-Cost VMs	Unoptimized	0.9139	0.9134	0.9134
	Optimized	0.9136	0.9153	0.9151
Avg. Isolation (%) for Low-Cost VMs	Unoptimized	0.1651	0.1379	0.1693
	Optimized	0.1220	0.0911	0.0917

Table 3: Impact of larger isolated LLC allocations for CS VMs on ST and LC VMs.

Nevertheless, the standard sharing and low-cost pressure are the determining factors in the amount of standard upsizing. To see this, we breakdown the cost function (Equation 7) into two dimensions in Figure 14. The X-axis shows the cost function component that decreases by upsizing standard LLC partitions, whereas the Y-axis shows the sum of other components, which increases with upsizing. Independent of the choice of standards, the Pareto-optimal allocation picked by the optimizer resides where the drawbacks of standard upsizing equal its benefits.

As a result of this versatility, CacheSlicer delivers similar LLC QoS to standards VMs for small and large settings. The $\frac{EWPC}{WPC}$ ratio is 98.1% and 97.9% for small and large standard settings; and the $\frac{IWPC}{WPC}$ ratio is 96.5% and 96.3%, respectively. In other words, the cost function ensures average delivery of ~98% of advertised LLC space, and more than 96% LLC isolation for standard VMs.

5.5.2 LLC demands of CS VMs and Percentage of CS VMs. To analyze how the mix and percentage of CS VMs affect the allocation optimization, we run the optimizer for Table 2 settings and the default 3.56% of CS VMs using the combined distribution of five production clusters. Table 3 summarizes the results of the optimization for these settings. As expected, larger CS VMs (CS sizes highlighted in blue) reduce isolation for standard VMs and lead to smaller WPC for low-cost VMs.

Figure 15(left) shows the impact of the percentage of CS VMs on the standard allocation optimization. Increasing CS percentage from 3.6% to 10.3% noticeably impacts the standard EWPC for both Small-Fast and Large-Slow settings,

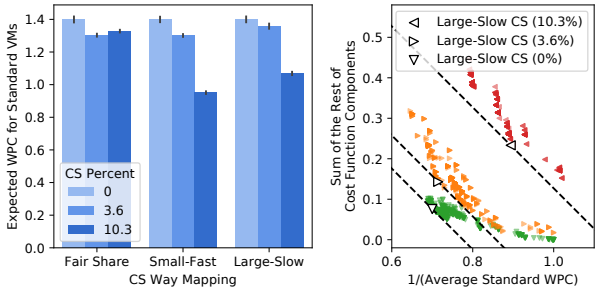


Figure 15: (L) Increasing CS VMs reduces the ST EWPC. (R) Breakdown of the optimization cost function components.

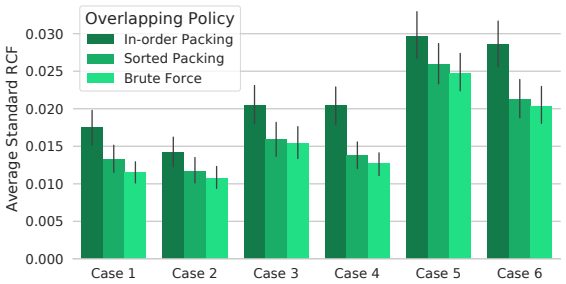


Figure 16: Comparing algorithms for server-level LLC sharing for standard VMs. Sorted sharing achieves comparable results to (unrealistic) brute force search.

while the Fair-Share CS way setting allows graceful upsizing of standards. Figure 15(right) shows the Pareto-optimal optimization curves for the Large-Slow case with varying the percentage of CS VMs. As the figure shows, upsizing standards to the same amount (same X value) has more overhead (higher Y value) when there are more CS VMs.

5.5.3 Server-Level Partial LLC Sharing Policies. So far, we have assumed the sorted packing algorithm for server-level LLC sharing for standard VMs. Figure 16 compares the average standard RCF for in-order packing, sorted packing, and an oracle via brute force search. The six cases in the figure correspond to different proportions and settings of CS VMs. The figure shows that using the simple sorted packing algorithm can improve the average standard RCF 12.5%-32.6%, generating comparable results to brute force. We also find that sorted packing is only ~12% slower than in-order packing (unrealistic brute force is ~4800x slower).

Overlapping the same number of ways translates into smaller shares for larger VMs. This is why sorted packing outperforms in-order packing. We illustrate this effect in Figure 8, where only larger VMs experience partial sharing.

5.6 Application-Level Performance

While CacheSlicer's cluster-level LLC management is evaluated in Section 5, we study how it can affect the end-to-end performance of cloud workloads. CacheSlicer is not about improving performance; instead, it targets delivering differentiated LLC QoS guarantees to public cloud users, and we

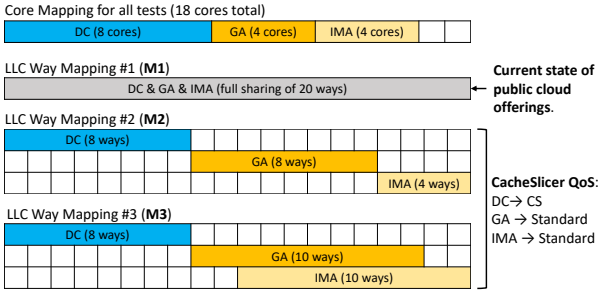


Figure 17: Comparing the effect of LLC allocations on application performance of three cloud workloads.

show that choosing different VM categories can improve performance, performance predictability, or cost.

We study co-location effects of three VMs running applications from the CloudSuite v3.0 benchmarks [20, 49]: Data Caching (DC), Graph Analytics (GA), and In-Memory Analytics (IMA). We use a production-class server (Intel Xeon E5-2686 v4 CPU [18 cores, 20-way 45MB LLC], 512GB main memory). All VMs run together: DC runs indefinitely and is set to deliver 225K requests per second (RPS) with 12 worker threads; GA (with 25GB of driver and 70GB of executor memory) and IMA VMs run iteratively. VMs use Ubuntu 16.04 LTS, and VM cores are pinned to server cores to reduce interference [19, 53, 68]. DC has 8 cores and 128GB of memory. GA and IMA have 4 cores and 112GB of memory, each.

Figure 17 shows the core allocation and three LLC mappings. Mapping #1 (M1) denotes the current state of cloud offerings where the entire LLC is shared. Mappings #2 and #3 (M2 and M3) correspond to CacheSlicer when the user selects CS category for the DC, and standard category for GA and IMA VMs. M3 offers larger LLC portions to standard VMs compared to M2. We summarize the results below.

DC's target QoS is P95 latency of 10ms [1], which is met when the user selects CS category guaranteeing LLC isolation for DC VM. As shown in Figure 18, the QoS violation of 16.4% in M1 (shared) is reduced to 4.9% and 0.8%, in M2 and M3, respectively. If the user's desired P95 latency for the same service rate were 20ms, the user would not need the isolated LLC share. This is why we advocate for exposing LLC QoS users to suit various demands and valuations.

Although GA and IMA have smaller LLC shares in M2 compared to M1, they can accommodate a co-located CS VM with relatively low impact on performance: GA and IMA experience only 2.3% and 4.3% slow-down, showing that the co-location of CS VMs is feasible.

Finally, moving from M2 to M3, GA's average performance remains almost the same, and IMA gains 2.7% speed-up. More importantly, as GA and IMA have overlapping LLC shares, GA and IMA experience 2.56x and 1.36x higher performance variations, respectively. These changes follow CacheSlicer's LLC QoS metrics: (1) lower sharing (RCF) to achieve less performance variation, and (2) higher shares (WPC) to achieve

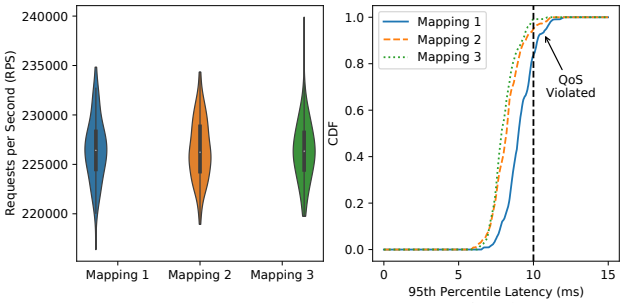


Figure 18: Delivering the same request rate (left), DC meets tail latency target in Mapping 2 & 3 (right).

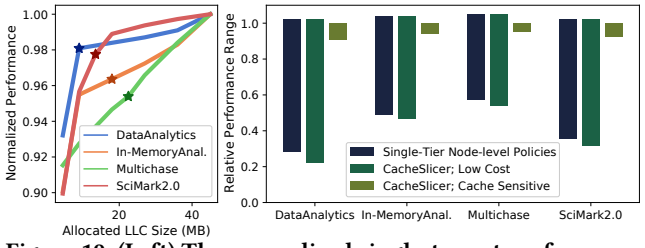


Figure 19: (Left) The normalized single-tenant performance of four workloads as a function of LLC size. The workloads' target performance is marked with stars. (Right) The performance variation range using different policies.

higher performance. Note that CacheSlicer's LLC-aware VM scheduler takes overlapping into account and reduces its possibility for standard VMs.

5.7 Comparison to Node-Level Partitioning

CacheSlicer is substantially different from node-level LLC management policies, making it hard to conduct a direct comparison. In CacheSlicer, the VM scheduler helps prevent excessive LLC contention at the node level with the appropriate VM categories. When the VM scheduler is LLC-agnostic, performance can degrade significantly even with the best node-level policy when too many tenants are mapped to a single node. Given such limitations, we situate CacheSlicer in comparison to node-level policies by experimentally comparing the best-case and worst-case co-location scenarios.

We use the same server as in Section 5.6 for this study. Since the types of cloud servers are known, a user can profile their application on the same server, locally or on bare-metal, to identify the application's LLC requirement. We do that for four different applications: (1) a Data Analytics workload from the CloudSuite v3.0 benchmark [20, 49], using six threads (3 cores); (2) an In-Memory Analytics workload from the CloudSuite v3.0 benchmark [20, 49], using eight threads (4 cores); (3) a dual-threaded (1 core) Multichase workload [3]; and (4) a single-threaded SciMark2.0 [56]. The results appear in the left sub-figure of Figure 19. For these applications, we choose various target performance and LLC allocation values (marked as stars in Figure 19). The question we seek to answer is: how would the performance vary compared to this target performance under different policies? We deliberately consider cases where co-tenants are replicas of the same workload. This is because if at least two dissimilar co-tenants run, one does better and the other does worse compared to their respective similar-co-tenant scenario.

Table 4 lists the best/worst-case scenarios for 3 policy groups. The first is single-tier node-level policies, which correspond to the majority of related work where LLC space is apportioned across workloads dynamically [18, 52, 58, 62, 72, 74, 76]. Although these policies have various profiling, classification, and partitioning mechanisms, they all behave

Workload	Single-Tier Node-level Policies		CacheSlicer; Low Cost (LC)		CacheSlicer; Cache Sensitive (CS)	
	best	worst	best	worst	best	worst
Data Analytics	tenant 1/1; full LLC	1/6 tenants; sharing entire LLC	tenant 1/1; full LLC	1/5 LC tenants sharing 1 way	tenant 1/1; assigned 4 isolated ways	1/5 tenants; 4 isolated ways
In-Memory Analytics	tenant 1/1; full LLC	1/4 tenants; sharing entire LLC	tenant 1/1; full LLC	1/3 LC tenants sharing 1 way	tenant 1/1; assigned 8 isolated ways	1/2 tenants; 8 isolated ways
Multichase	tenant 1/1; full LLC	1/18 tenants; sharing entire LLC	tenant 1/1; full LLC	1/17 LC tenants sharing 1 way	tenant 1/1; assigned 10 isolated ways	1/2 tenants; 10 isolated ways
SciMark2.0	tenant 1/1; full LLC	1/36 tenants; sharing entire LLC	tenant 1/1; full LLC	1/35 LC tenants; sharing 1 way	tenant 1/1; assigned 6 isolated ways	1/3 tenants; 6 isolated ways

Table 4: Description of different co-locations scenarios delivering best- and worst-case performance.

similarly in extreme cases corresponding to best- and worst-case performance. In the best-case, a workload with no co-tenant is assigned the entire LLC as it ensures the best performance (lowest slowdown). In the worst-case, replicas of the same workload could either share the entire LLC or fall into smaller partitions. The former case delivers better performance due to higher associativity and will be chosen by node-level policies. Note that since all competing workloads are the same in this worst-case scenario, memory bandwidth partitioning [51] provides no added value. Similarly, methods identifying latency-critical workloads [55] cannot help as all replicas fall into the same tier (latency-critical or batch).

The second policy is Low Cost (LC) VMs with CacheSlicer. Their best-case performance behavior is similar to the previous policy group, as an LC VM utilizes the entire LLC when possible. On the other hand, since CacheSlicer has three tiers, the worst-case scenario happens when a Cache Sensitive (CS) VM with a massive LLC allocation is co-located with many LC replicas that have to share just one LLC way. The numbers of LC co-tenants we consider are listed in Table 4. Finally, the third policy is a CS VM with CacheSlicer. It produces the same LLC allocation size under best and worst-case scenarios. Since CacheSlicer considers LLC a hard resource for CS workloads, it tends to produce lower co-tenant counts.

The performance ranges for these cases are shown in the right of Figure 19. The top of each bar corresponds to the best performance and the bottom corresponds to the worst, normalized against the target performance for each application. Node-level policies and LC VMs can deliver better performance compared to the target, as the workload can consume the entire LLC if it has no co-tenant. However, both policy types can suffer a huge slowdown if the scheduler assigns too many co-tenants to their node. The degree of

this slowdown is a function of application and number of cotenants. Note that LC VMs do worse than node-level policies, as they can get choked by CS VMs. In contrast, the figure shows that the CS VMs deliver performance consistency as expected. The slight slowdown is due to other shared resources, such as memory bandwidth. The main conclusion is that consistent performance needs both intentional allocation of cache ways and LLC-awareness by the VM cluster scheduler; state-of-the-art node-level policies have not targeted these characteristics.

6 FREQUENTLY ASKED QUESTIONS

How does CacheSlicer guarantee performance for latency sensitive workloads with just considering LLC? CacheSlicer does not provide performance guarantees for latency sensitive workloads. Instead it targets delivering LLC QoS guarantees to cache sensitive workloads. We discuss the difference between these two in Section 4.1. This was an informed decision as unlike on-premise or private environments, public cloud providers have no means to correctly interpret the performance of opaque VMs and no information on the performance valuation. Recognizing this context is vital for understanding design choices in this work.

How about the performance effects of decreased associativity? Way partitioning can hurt cache capacity super linearly due to decreased associativity [35, 71]. CacheSlicer handles this by providing transparency and consistency. We expose the cache allocation parameters to users – particularly the number of LLC ways. Moreover, we guarantee providing it to them consistently no matter which server their VM lands on. These two factors enable users to decide if the reduced performance is acceptable to them. Also note that for standard VMs we ensure a minimum WPC of 1, which means linearly higher associativity with increased VM size. This guarantees associativity diversity.

How about the added complexity for the users? For ordinary users, the standard category will be chosen by default. On the other hand, this knob will provide more control for users who want more control. This is similar to the current state of cloud configurations where users can go to the "advanced settings panel" to fine-tune extra options.

7 RELATED WORK

CacheSlicer is the first scheme to offer **cluster-level** LLC allocations for **public clouds**. Table 5 compares it to the related work on way partitioning/clustering. For LLC partitioning techniques beyond way partitioning, we refer the reader to related survey studies [24, 45]. Note that several early studies on cache contention mitigation in cloud relied on scheduling solutions without LLC partitioning [5, 12, 67].

Most systems using way partitioning rely on performance profiling. Such reliance limits their use for public clouds VMs due to 1) discrepancy of external performance metrics and application performance [22], 2) lack of knowledge on user valuation of performance, 3) adversarial exploitations, and 4) non-trivial profiling overhead [81]. Ginseng [22] showed that game theory principles can mitigate the need for profiling. CacheSlicer eliminates reliance on profiling by delivering LLC QoS guarantees and letting the users decide.

Related Work	Goal	Scope	QoS Level(s)	Profiling Free	Cluster Level
Heracles [42]	LS co-location	task	2 (LS/not)	✗	✗
CATalyst [39]	side channel defense	VM	2 (secure/not)	✗	✗
Dirigent [82]	LS co-location	task	2 (LS/not)	✗	✗
Ginseng [22]	IaaS auction-based	VM	-	✓	✗
SWAP [72]	finer partitioning	task	-	✗	✗
Selfa [62]	better system fairness	task	-	✗	✗
KPart [18]	better avg. slowdown	task	-	✗	✗
Sprabery [65]	side channel defense	cont/VM	2 (secure/not)	✓	✗
DCAPS [74]	dynamic overlapping	task	-	✗	✗
dCAT [76]	dynamic allocation	task	-	✗	✗
SDCP [28]	side channel defense	cont	2 (secure/not)	✗	✗
Pons [55]	lower turnaround time	task	2 (llc-critical/not)	✗	✗
CoPart [51]	joint LLC & Mem. BW	task	-	✗	✗
LFOC [23]	OS support for fairness	task	3	✗	✗
CLITE [52]	multiple LS & BG tasks	task	2 (llc-critical/BG)	✗	✗
Rhythm [81]	breaking down LCs	task	2 (LC/BE)	✗	✗
SATORI [58]	fairness + throughput	task	-	✗	✗
CacheSlicer	cloud multi-QoS	VM	flexible (Def. 3)	✓	✓

Table 5: The taxonomy of prominent LLC management systems using way partitioning sorted by publication date.

Some systems use way partitioning to improve execution of all applications [18, 54, 58, 62, 72, 74]. These systems are complementary to CacheSlicer as it targets public clouds, provides QoS at VM granularity, and enforces LLC policies at the cluster level with an LLC-aware scheduler. A few systems provide isolation for CS workloads [42, 82]. Similarly, CacheSlicer delivers full isolation to CS workloads, however, at the VM level. Finally, some studies have used way partitioning to mitigate LLC side-channel attacks [17, 26, 35, 39, 40, 65, 77]. We did not study the security implications of CacheSlicer.

8 CONCLUSION

In this paper, we quantified LLC competition among VMs in Azure and introduced CacheSlicer, the first system providing cluster-level support for LLC management for public clouds. CacheSlicer enables multiple levels of LLC QoS, providing guaranteed LLC isolation for cache-sensitive VMs, best-effort isolation for standard VMs, and no isolation for low-cost VMs. CacheSlicer uses an offline policy optimizer as well as an LLC-aware VM cluster scheduler. Our evaluation showed that cluster support and policy optimization improve LLC allocation, and CacheSlicer successfully meets the LLC requirements of the differentiated VM categories.

ACKNOWLEDGMENTS

We thank Behnaz Arzani, Alexey Lavrov, Tri Nguyen, Ila Nimgaonkar, and David Wentzlaff for constructive discussions and their valuable feedback on earlier drafts of this paper. We also thank the anonymous reviewers and our shepherd, Srinivas Narayana, for helping us improve the paper.

REFERENCES

[1] [n.d.]. CloudSuite: Data Caching. https://github.com/parsa-epfl/cloudsuite/blob/master/docs/benchmarks/data-caching.md. Accessed: 2021-5-28.

[2] [n.d.]. Intel® RDT Software Package. https://github.com/intel/intel-cmt-cat. Accessed: 2021-5-28.

[3] [n.d.]. Multichase - a pointer chaser benchmark. https://github.com/google/multichase. Accessed: 2021-5-28.

[4] 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developers-manual.pdf. Accessed: 2021-5-28.

[5] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young-ri Choi, and Jaehyuk Huh. 2012. Dynamic virtual machine scheduling in clouds for architectural shared resources. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*.

[6] David H Albonesi. 1999. Selective cache ways: On-demand cache resource allocation. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*.

[7] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for resource-harvesting VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[8] Amitabha Banerjee, Rishi Mehta, and Zach Shen. 2015. NUMA aware I/O in virtualized systems. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*.

[9] Anton Beloglazov and Rajkumar Buyya. [n.d.]. Energy Efficient Resource Management in Virtualized Cloud Data Centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE Computer Society.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*.

[11] Jichuan Chang and Gurindar S. Sohi. 2014. Cooperative Cache Partitioning for Chip Multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*.

[12] Liuhua Chen, Haiying Shen, and Stephen Platt. 2016. Cache contention aware virtual machine placement and migration in cloud datacenters. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*.

[13] Quan Chen, Shuai Xue, Shang Zhao, Shanpei Chen, Zhuo Song, Yihao Wu, Yu Xu, Tao Ma, Yong Yang, and Minyi Guo. 2020. Alita: comprehensive performance isolation through bias resource management for public clouds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*.

[14] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*.

[15] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and José F. Martínez. 2017. Workload Characterization of Interactive Cloud Services on Big and Small Server Platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*.

[16] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*.

[17] Christina Delimitrou and Christos Kozyrakis. 2017. Bolt: I Know What You Did Last Summer... In The Cloud. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[18] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*.

[19] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*.

[20] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[22] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2016. Ginseng: Market-Driven LLC Allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association.

[23] Adrian Garcia-Garcia, Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. 2019. LFOC: A Lightweight Fairness-Oriented Cache Clustering Policy for Commodity Multicores. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*.

[24] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–36.

[25] Aristide Grange, Imed Kacem, and Sébastien Martin. 2018. Algorithms for the bin packing problem with overlapping items. *Computers & Industrial Engineering* 115 (2018), 331–341.

[26] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.. In *USENIX Security Symposium*.

[27] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. 2020. Protean: VM Allocation Service at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[28] Myeonggyun Han, Seongdae Yu, and Woongki Baek. 2018. Secure and Dynamic Core and Cache Partitioning for Safe and Efficient Server Consolidation. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*.

[29] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17.

[30] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*.

[31] Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li. 2009. A simple cache partitioning approach in a virtualized environment. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*.

[32] Ram Srivatsa Kannan, Animesh Jain, Michael A. Laurenzano, Lingjia Tang, and Jason Mars. 2018. Proctor: Detecting and Investigating Interference in Shared Datacenters. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[33] H. Kim, A. Kandhalu, and R. Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *2013 25th Euromicro Conference on Real-Time Systems*.

[34] Yeseong Kim, Ankit More, Emily Shriver, and Tajana Rosing. 2019. Application performance prediction and optimization under cache allocation technology. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[35] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51 '18)*.

[36] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680.

[37] Shivam Kundan and Iraklis Anagnostopoulos. 2021. Priority-Aware Scheduling Under Shared-Resource Contention on Chip Multicore Processors. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*.

[38] Ankur Limaye and Tosiron Adegbija. 2018. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[39] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*.

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*.

[41] Ming Liu and Tao Li. 2014. Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.

[42] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[43] Jason Mars, Lingjia Tang, and Mary Lou Soffa. 2011. Directly Characterizing Cross Core Interference Through Contention Synthesis. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC '11)*.

[44] Milo MK. Martin, Mark D. Hill, and Daniel J. Sorin. 2012. Why on-chip cache coherence is here to stay. *Commun. ACM* 55, 7 (2012), 78–89.

[45] Sparsh Mittal. 2017. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.* 50, 2, Article 27 (May 2017), 39 pages.

[46] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-Clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*.

[47] S. Noll, J. Teubner, N. May, and A. Böhm. 2018. Accelerating Concurrent Workloads with CPU Cache Partitioning. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*.

[48] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX.

[49] Tapti Palit, Yongming Shen, and Michael Ferdman. 2016. Demystifying cloud benchmarking. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*.

[50] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. 2017. Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*.

[51] Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*.

[52] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[53] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18), Renton, WA*.

[54] Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, and Julio Pons. 2020. Phase-Aware Cache Partitioning to Target Both Turnaround Time and System Performance. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2556–2568.

[55] Lucia Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, and Julio Pons. 2018. Improving System Turnaround Time with Intel CAT by Identifying LLC Critical Applications. In *Euro-Par 2018: Parallel Processing*. Springer International Publishing.

[56] Roldan Pozo and Bruce Miller. [n.d.]. SciMark 2.0. https://math.nist.gov/scimark2/. Accessed: 2021-5-28.

[57] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. 2000. Reconfigurable Caches and Their Application to Media Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*.

[58] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*.

[59] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*.

[60] Daniel Sanchez and Christos Kozyrakis. 2012. Scalable and efficient fine-grained cache partitioning with Vantage. *IEEE Micro* 32, 3 (2012), 26–37.

[61] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. 2016. A Software Cache Partitioning System for Hash-Based Caches. *ACM Trans. Archit. Code Optim.* 13, 4, Article 57 (Dec. 2016), 24 pages.

[62] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María E. Gómez. 2017. Application clustering policies to address system fairness with Intel's Cache Allocation Technology. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*.

[63] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*.

[64] Mohammad Shahrad, Cristian Klein, Liang Zheng, Mung Chiang, Erik Elmroth, and David Wentzlaff. 2017. Incentivizing Self-capping to Increase Cloud Utilization. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*.

[65] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. 2018. Scheduling, Isolation, and Cache

Allocation: A Side-Channel Defense. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*.

[66] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*.

[67] Alain Tchana, Bao Bui, Boris Teabe, Vlad Nitu, and Daniel Hagimont. 2016. Mitigating performance unpredictability in the IaaS using the Kyoto principle. In *Proceedings of the 17th International Middleware Conference*.

[68] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M Swift. 2012. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM conference on Computer and communications security*.

[69] Akshat Verma, Puneet Ahuja, and Anindya Neogi. 2008. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*. Springer-Verlag New York, Inc.

[70] Po-Han Wang, Cheng-Hsuan Li, and Chia-Lin Yang. 2016. Latency Sensitivity-based Cache Partitioning for Heterogeneous Multi-core Architecture. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*.

[71] Ruisheng Wang and Lizhong Chen. 2014. Futility Scaling: High-Associativity Cache Partitioning. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

[72] Xiaodong Wang, Shuang Chen, Jeff Setter, and José F. Martínez. 2017. SWAP: Effective fine-grain management of shared last-level caches with minimum hardware support. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*.

[73] Barry Wolford, Thomas Speier, and Dileep Bhandarkar. 2017. Qualcomm Centriq™ 2400 Processor. https://www.qualcomm.com/media/documents/files/qualcomm-centriq-2400-processor.pdf.

[74] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[75] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*.

[76] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*.

[77] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*.

[78] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A Dynamic Cache Partitioning System Using Page Coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*.

[79] Lingfang Zeng, Yang Wang, Wei Shi, and Dan Feng. 2013. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *2013 International Conference on Cloud Computing and Big Data*.

[80] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards Practical Page Coloring-based Multicore Cache Management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*.

[81] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. 2020. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*.

[82] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*.

[83] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.