dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-Service

Cong Xu
IBM Research Austin
xucong@us.ibm.com

Karthick Rajamani
IBM Research Austin
karthick@us.ibm.com

Alexandre Ferreira
IBM Research Austin
apferrei@us.ibm.com

Wesley Felter
IBM Research Austin
wesley@felter.org

Juan Rubio
IBM Research Austin
rubioj@us.ibm.com

Yang Li*
Carnegie Mellon University
jerryyangli@gmail.com

ABSTRACT

In the modern multi-tenant cloud, resource sharing increases utilization but causes performance interference between tenants. More generally, performance isolation is also relevant in any multi-workload scenario involving shared resources. Last level cache (LLC) on processors is shared by all CPU cores in x86, thus the cloud tenants inevitably suffer from the cache flush by their *noisy neighbors* running on the same socket. Intel Cache Allocation Technology (CAT) provides a mechanism to assign cache ways to cores to enable cache isolation, but its static configuration can result in underutilized cache when a workload cannot benefit from its allocated cache capacity, and/or lead to sub-optimal performance for workloads that do not have enough assigned capacity to fit their working set.

In this work, we propose a new dynamic cache management technology (dCat) to provide strong cache isolation with better performance. For each workload, we target a consistent, minimum performance bound irrespective of others on the socket and dependent only on its *rightful* share of the LLC capacity. In addition, when there is spare capacity on the socket, or when some workloads are not obtaining beneficial performance from their cache allocation, dCat dynamically reallocates cache space to cache-intensive workloads. We have implemented dCat in Linux on top of CAT to dynamically adjust cache mappings. dCat requires no modifications to applications so that it can be applied to all cloud workloads. Based on our evaluation, we see an average of 25% improvement over shared cache and 15.7% over static CAT for selected, memory intensive, SPEC CPU2006 workloads. For typical cloud workloads, with Redis we see 57.6% improvement (over shared LLC) and 26.6% improvement (over static partition) and with ElasticSearch we see 11.9% improvement over both.

ACM Reference Format:
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 *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3190508.3190555

*The work was done during Yang's internship at IBM.

1 INTRODUCTION

Public clouds have grown significantly in the last few years and the Infrastructure-as-a-Service (IaaS) segment's adoption is expected to grow even more rapidly in the next few years [6]. The adoption of *virtual servers* enabled with virtualization technologies has allowed companies such as Amazon and Google with vast internal needs for computing to offer that same physical infrastructure they use to the public as IaaS. Sharing of resources among different workloads and tenants allows better utilization of the physical infrastructure, lowering the cost of computing for all. However, multiple workloads on same server can lead to performance interference due to competition for the shared resources. Performance isolation is important for application availability (e.g. protect against Denial-of-Service attacks) as well as for responsiveness and predictability (e.g. for capacity planning). Researchers have pointed out the importance of resource isolation in order to have performance isolation [12, 13] for both responsiveness of latency-critical applications as well as to reduce the dominating tail-latency impact for applications built with microservices model. Public clouds today offer *dedicated instances* (virtual server that has a physical host all to itself) as a means to get consistent performance. However, this reduces overall utilization of infrastructure and drives up the cost of computing. The key to address this dichotomy is leveraging the ability to provide performance isolation within a server when needed while not giving up on the ability to share/re-distribute physical resources among multiple workloads.

Providing dedicated resources to a workload within a server not shared with other workloads on that same server provides a good start for balancing isolation and efficiency needs. That is possible for certain resources such as processor cores, memory capacity, and I/O bandwidth where the hardware, operating systems and hypervisors have mature mechanisms to partition these resources. However, other shared resources such as CPU cache are currently not controlled intelligently. These resources are fairly precious relative to the cores and inherently designed to be shared by all workloads running on a processor socket to maximize their utilization.

While today IaaS's focus is on just availability of contracted resources in the form of a virtual server, we envision a next-generation, *performance-sensitive IaaS* that would ensure consistent performance for any workload using those contracted resources with resource isolation mechanisms and drive up utilization of resources by also enabling resource sharing without impacting performance consistency. To do this, we cannot just statically partition the resources among different workloads (that would lead to fragmentation and under-utilization) but need to do so dynamically in a performance-sensitive manner.

In this work, we examine how we can do this effectively for the processor's shared last-level cache (LLC). Intel x86 processors have an LLC that is set-associative, inclusive and shared by all the cores on the processor socket. Recognizing the need for cache isolation, Intel introduced a *Cache Allocation Technology* (CAT) [20] that allows runtime partitioning of the L3 cache among different cores. CAT works by restricting which ways a core can evict cache lines from, in practice limiting the size and associativity of cache each core is able to use.[1] However, we observe that CAT causes sub-optimal performance for some memory-intensive applications when used naively. CAT is a low-level, hardware mechanism (and drivers) which enables cache partitioning. It needs to be supplemented with a controller managing it to dynamically change the size of cache partitions based on runtime behavior.[2] Since public clouds generally treat workloads as black boxes, they cannot estimate the cache needs of workloads in advance. Many cloud users are also not sophisticated enough to know how much cache capacity their workloads need either. If a workload is assigned a cache allocation smaller than its working set, it will suffer *expensive* cache misses. Too-large cache allocations leave the cache poorly utilized, preventing other workloads from getting as much cache capacity as they could benefit from.

We propose a dynamic cache allocation technology (dCat) built on top of CAT. dCat first targets a *guaranteed minimum* performance level for a workload, irrespective of "noisy neighbor" workloads running on the same socket — this performance is what can be attained by using CAT for cache based isolation. This guaranteed minimum level termed *baseline* offers a consistent baseline performance level that can be leveraged for planning purposes. In addition, it enables higher performance for workloads that can use more cache capacity when others are not getting much benefit from their assigned LLC. dCat optimizes the cache assignment at running time and always allocates cache to those workloads that really need it, while still guarantees that all workloads at least get the same performance that they would have under static cache partitioning. We implemented dCat as a Linux daemon which is transparent to the workloads (and VMs, containers) running on the x86 platform. It can be applied to all existing cloud environments without any modification to both the OS kernel and the upper-level applications.

To summarize, our contributions in this paper are:

- We establish the notion of a performance-sensitive IaaS framework where a consistent baseline performance level can be guaranteed based on contracted resource allocation for a workload while allowing dynamic resource adjustments to improve physical resource utilization and cloud efficiency.

[1] In this paper we do not consider cache isolation between hyperthreads of the same core, since those threads have unavoidable interference caused by shared execution resources.
[2] Intel cache monitoring technology (CMT) cannot substitute such controller, as CMT has no means to determine the optimal cache partitioning.

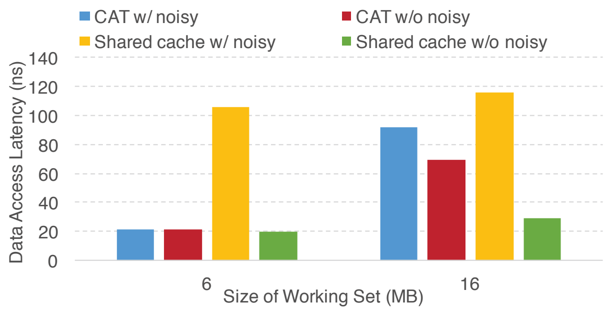


Figure 1: Impact of cache interference for MLR

- We explore the cache needs of different workloads with diverse working set sizes and data access patterns in the multi-tenant cloud environment.
- We develop dCat in real systems, which provides dynamic cache allocation to different workloads according to their cache needs.
- Our evaluations with the prototype of dCat show that it not only effectively guarantees a minimum performance to workloads irrespective of noisy neighbor presence, but also leads to little overhead and improves performance by up to 57.6% over an unmanaged shared cache and by up to 26.6% over a statically-partitioned cache across a range of different workloads.

2 CHALLENGES

The LLC in Intel x86 processors is inclusive[3], set-associative and shared among all cores in a processor socket. We look at some of the main challenges in managing this cache for simultaneously providing multi-tenant performance isolation and increasing utilization in this section.

2.1 The Impact of Cache Interference

We use two simple, well-understood, internally developed memory access benchmarks for this exercise: MLR and MLOAD. MLR is a stream of random read accesses to an array, while MLOAD is a stream of sequential read accesses to an array. For each of the workloads, the size of its array determines the size of its working set.

Figure 1 shows the performance of MLR (latency, lower is better) with 6MB working set and 16MB working set. We run four scenarios for each working set: without *noisy neighbors*, with *noisy neighbor* workloads which are two instances of MLOAD (60MB working set), and both with CAT used to assign 13.5MB of dedicated cache (6 LLC cache-ways out of a Xeon-E5 processor with a 20-cache-way 45MB LLC) to MLR and with fully shared cache (*i.e.* without CAT). The working sets are sized so that performance is largely dependent on how well the workload can utilize the LLC. For the MLR-6MB, we can see that its performance will be significantly hurt by the noisy neighbors if we do not provide any dedicated cache (compare Shared cache w/noisy with Shared cache w/o noisy); however, if we provide

[3] The inclusivity implies that data kicked out of the LLC because of capacity or conflict miss is also dropped from the closer caches (L1, L2).

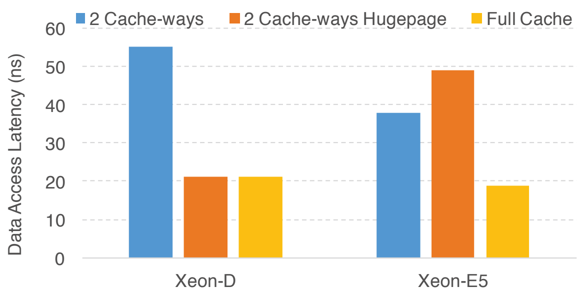


Figure 2: Impact of CAT-limited cache size

the 13.5MB dedicated cache, its performance is protected and isolated from the neighbors (compare CAT w/noisy with Shared cache w/o noisy). This demonstrates that CAT can provide performance isolation if the dedicated cache for a workload is large enough to hold its working set. However, if the dedicated cache is not large enough, CAT will fail to provide performance isolation. For example, though 13.5MB dedicated LLC is provided, the MLR-16MB is severely interfered by its noisy neighbors (compare CAT w/noisy with Shared cache w/o noisy), because in this case, the MLR-16MB has to frequently load data from memory instead of only from LLC. Therefore, we can conclude that whether CAT can provide performance isolation between workloads is highly dependent on the size of the dedicated cache it statically allocates to each workload run by different tenants.

However, determining the size for a dedicated cache for each workload is not a trivial task. In order to determine the appropriate size, we need to first predict the working set of a workload. However, in the majority of cases, cloud providers have very limited beforehand knowledge about tenants' workloads (or their working set size and how they change over time). In addition, even if we can accurately predict the working set of a workload, it is also hard to determine an appropriate cache size to hold its working set because of cache *conflict* misses [10]. Conflict miss is a specific cache miss that may happen when multiple cache lines compete for the associative ways in a single set of the cache. This conflict miss exists even the huge page (2MB on Linux x86) is enabled for workloads.

This is exemplified in Figure 2, where we show average access latencies of MLR on two Broadwell systems (8-core, 12-way 12MB LLC Xeon-D and 18-core 20-way 45MB LLC Xeon-E5). CAT is used to restrict the cache allocation for the blue bars (4KB regular page) and orange bars (2MB huge page). The yellow bars represent performance for full cache capacity (4KB regular page). CAT reduces capacity by limiting the number of ways that a core can access. Enough capacity is provided even under the reduced cache size to not have capacity misses (2MB workload working set with 2-way 2MB dedicated LLC for Xeon-D, and 4.5MB workload working set with 2-way 4.5MB dedicated LLC for Xeon-E5). However, as can be seen, the performance is still significantly lower with the reduced cache size. This is because the reduced capacity also comes with reduced associativity which results in higher conflict misses leading to higher average access latency. With the virtual to physical address mappings, a contiguous virtual address space larger than a page size (4KB) does not necessarily occupy a contiguous physical address space. *Mapping to cache lines is determined by the physical address and whether a memory location finds place within restricted cache ways depends on both the mapping (potentially conflicting for cache lines) as well as total space in those ways, and not just the latter*. When associativity is reduced (by limiting cache ways), the impact of conflict from non-contiguous address mapping is seen more. Huge page can guarantee more contiguous physical address after translation, this is why the 2MB working set workload using huge page achieves similar performance to the full cache on Xeon-D. However, the workload having larger working set than one huge page (*e.g.* 4.5MB working set on Xeon-E5) still suffer the conflict miss. Figure 3 shows the corresponding histogram of the number of cache lines that are mapped to each cache set. We observe that, with 4KB page, even when 2 cache ways of LLC are allocated to equal the working set size, conflict misses can still occur as in the case of Xeon-D around 32.5% of sets have 3 or more cache lines mapped to (about 29% of sets for Xeon-E5); and therefore, these cache lines will conflict/compete for the 2 allocated cache ways. In Xeon-D hugepage case, 2MB working set only needs 1 huge page to fit so that it just can be perfectly cached by the allocated LLC leading to zero conflict miss. Unfortunately, in Xeon-E5 hugepage case, 4.5MB working set needs 3 pages but the cache can only fully hold 2. After address translation, we still see about 11.2% of sets have 3 cache lines mapped to thus resulting in significant conflict misses. These conflict misses make it harder to predict the cache needs for a workload even if we know its working set size.

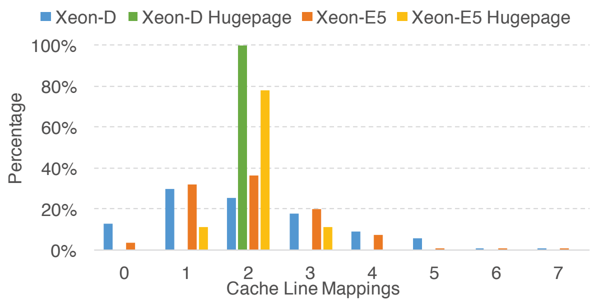


Figure 3: Cache set conflicts on Intel Broadwell Processors.

CAT provides the means to create cache usage isolation between cores (and so between different workloads running on different cores), but by doing so it limits cache capacity as well as associativity available to a specific workload to a static value that is hard to appropriately determine beforehand. Consequently, an adaptive approach dCat is proposed to manage the cache in this work. dCat operates based on runtime behavior and allocates additional cache to those workloads that could benefit from them, while still guaranteeing that all workloads get at least their baseline performance level.

2.2 Problems of Alternative Solutions

We have demonstrated above the static LLC allocation provided by Intel CAT leaves significant system performance on the table. Next, we examine several alternative solutions and argue their shortcomings when used in real systems.

OS-level page coloring Lin's study [29] aims to offer cache partitioning by using OS-level page coloring which directly assigns LLC among threads in the real system. All application threads are classified into one of four classes based on their performance degradation using only 1MB LLC compared to the baseline configuration using 4MB. Although this color-based classification scheme may be applicable to workload creation, it is unable to be easily used for dynamic, on-the-fly classification of workloads behavior. In particular, measuring the performance degradation requires running two copies of a program on two cores simultaneously, each with its own dedicated LLC. Due to its limitation, it has not been accepted by current Linux kernel.

Fine-grained cache partitioning on chip Some researchers have noticed the performance interference caused by shared LLC and tried to provide a series of chip-level cache allocation mechanisms according to the workloads behaviors [22, 23, 27, 36, 37, 41, 42]. These approaches are explored in simulation as opposed to our proposal which is implemented in real systems. All these prior proposals require additional hardware support for the cache partitioning capability. On the contrary, our proposal dCat is a software mechanism leveraging the currently available hardware knobs (performance counters and Intel CAT) to perform performance driven cache partitioning with performance isolation. It does not require any hardware modification for the current hardware platform, making it easier to be deployed. Besides, these prior proposals can not be directly applied to our goals of maintaining performance isolation first and then maximizing system throughput. The key differences in our work compared with prior proposals start with our goal of performance isolation in multi-tenant/-workload situations. A baseline performance is guaranteed for each workload. Cache utilization and resulting performance improvement are structured on top of this guarantee. Prior works mostly focus on different dynamic cache partitioning approaches in multiple workloads context for improving overall system miss-rate/performance (not performance isolation).

3 DESIGN

Our premise is that a dynamic cache allocation mechanism for the shared LLC can improve the performance of memory intensive workloads as well as guarantee the performance isolation among cloud workloads. dCat achieves this by dynamically adjusting the cache allocation to different workloads according to their behaviors. If some workloads do not use LLC or are unable to benefit from current LLC allocation, dCat reduces their cache size and reassign it to those who take advantage of a larger cache size to get better performance. The cache is allocated in terms of number of ways, which is the minimum partition unit of Intel Xeon processors. In dCat, one of the objectives is never to impact the performance of the workloads (when compared to the expected performance while using the reserved cache size). To realize this expectation, dCat performs five steps: *Get Baseline, Collect Statistics, Detect Phase Change, Categorize Workloads* and *Allocate Cache* which are shown in Figure 4.

As used in the present specification, the term "workload" refers to an application run by a tenant. It can also apply to a virtual machine (VM) or a container owned by a tenant in public or private cloud. If the "workload" uses more than one CPU core, we conduct the following five steps based on the overall performance of all involved cores. This means if one of the applications run by a tenant in a VM is cache-sensitive, this VM is considered to benefit from increased cache.

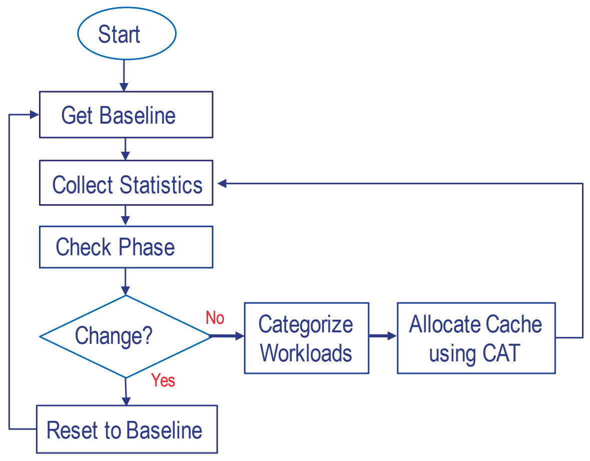


Figure 4: Flow chart of dCat.

3.1 Get Baseline

The first step in dCat is to determine the baseline performance for each workload. Baseline performance is defined as the performance of the workload when using the predefined cache partition (e.g. the cache partition paid by tenants). Since one workload can behave very differently in terms of cache needs over time, phase detection is essential. Baseline performance is only defined for a specific workload phase, so it has to be determined again at every phase change. In general terms, dCat tries to determine if a workload would benefit, or be indifferent to suffering if more or less cache is made available to it. The workload classification, that is specific for each phase and current cache size, is done by either inferring behavior over measured metrics or by comparing performance when different cache sizes are used.

Baseline performance for dCat is identified as the measured IPC for this workload phase when running with the statically predefined cache size (number of ways reserved). IPC is generally accepted as a good indicator of relative workload performance [26] and it is adversely affected by increase of cache misses.

3.2 Collect Statistics

The cache workload behavior has to be determined so dCat can allocate cache effectively. Unfortunately, in the cloud environment, each workload runs as a black box preventing dCat to have any pre-existing information about it. To mitigate this issue, we periodically collect the following information of each workload which is used by the *phase change detection, workload categorization*, and *cache allocation* steps:

- L1 cache references ($l1\_ref$): The total L1 references of all cores assigned to a workload. In Intel commodity processor, all data accesses go to the L1 cache first. Thus we use it to estimate the amount of LOAD/STORE instructions issued by this workload.

- LLC references (llc_ref): The total LLC references of all cores assigned to a workload. Low llc_ref means this workload does not require lots of LLC thus can not benefit from it.
- LLC cache misses (llc_miss): The total LLC cache misses suffered by a workload. We calculate the LLC cache miss rate (llc_miss_rate) by llc_miss/llc_ref.
- Retired instructions (ret_ins): Instructions retired by all cores assigned to a workload.
- IPC: The IPC of a workload which is the ratio of the retired instructions to the unhalted cycles of all core assigned to a workload.

We also define some corresponding thresholds to help us determine the workload characteristic in memory access: L1 cache reference threshold ($l1\_ref\_thr$), LLC reference threshold (llc_ref_thr), LLC cache miss threshold ($llc\_miss\_rate\_thr$), etc. All these thresholds are configurable depending on the needs of users.

dCat collects metrics per core. For those workloads using multiple CPU cores, dCat measures the performance of all used cores and calculate the average to evaluate the workloads.

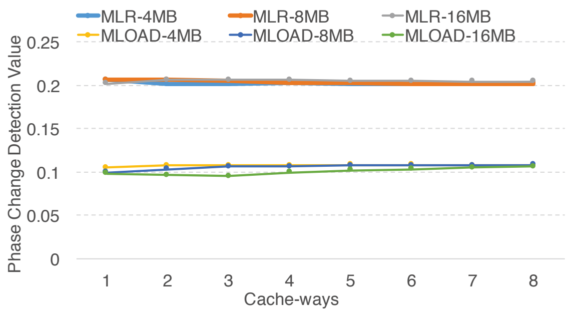


Figure 5: The phase change detector of dCat.

3.3 Detect Phase Change

Since IPC is used to determine workload performance and IPC also varies per workload phase, any workload phase change invalidates the performance comparison by changing the *baseline* IPC. dCat uses a simplified phase detection algorithm based on measuring the number of memory accesses (*i.e.* LOAD and STORE) per instruction (estimated by $l1\_ref/ret\_ins$) and assuming that sizable changes (10% used as the threshold in our prototype) on this value represent a phase change. We observe that the value of memory accesses per instruction is only related to the workload itself (*e.g.*internal logic) and is independent of the memory subsystem configuration (*e.g.* cache allocation). This is verified by a simple experiment in which we run the MLR and MLOAD with different working set size and varied cache-ways from 1 to 8. The results can be found in Figure 5. This is a very simple phase change detection but it meets our need of being reasonably independent of IPC and works well in our tests. There are other workload phase change detection methods [8, 9, 38] that can be used and they are pluggable into our work.

3.4 Categorize Workloads

With this baseline performance level determined and the collected statistics, the workloads can be categorized into the following classes based on cache utilization. That is, it can be determined 1) which cache allocations are being underutilized or over utilized, 2) which workloads could benefit from an increased amount of cache, and/or 3) which workloads would not be harmed by a reduced cache size.

- Reclaim: It is a state when a phase change is detected and the workload has to return to the baseline state.
- Receiver: It is a receiver if it benefits from more cache but suffers from less cache.
- Donor: It is classified as donor if the workload does not suffer if cache is reduced and also does not benefit from more cache.
- Keeper: It is a keeper if it would suffer if less cache is available but does not benefit from more.
- Streaming: Streaming is a special classification for workloads that even though they have a lot of cache misses, there is no reuse of data in the cache.
- Unknown: The unknown classification is used when no determination can be made using the current information. In dCat, this indicates that this workload at this size can be a *Receiver*, *Donor* or *Streaming* but a determination requires a comparison between different cache sizes. Some workloads with cyclic access pattern [35] have this behavior.

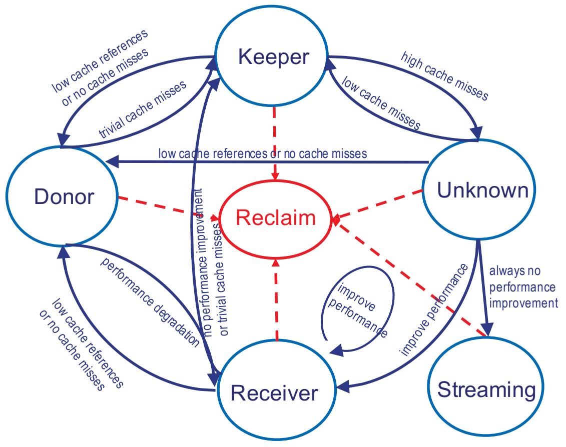
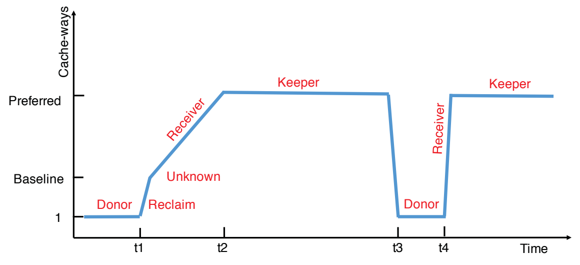

Keeper
Donor
Reclaim
Unknown
Receiver
Streaming
low cache references or no cache misses
trivial cache misses
high cache misses
low cache misses
low cache references or no cache misses
no performance improvement or trivial cache misses
performance degradation
low cache references or no cache misses
improve performance
improve performance
always no performance improvement

Figure 6: State Transition Diagram of dCat.

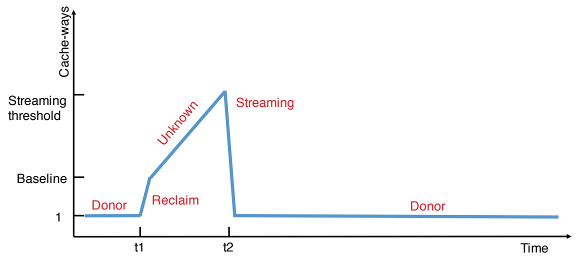
The relationship among these categories and their transition are displayed in Figure 6. The start state for any workload is *Keeper*. If this workload is idle or has low utilization of LLC (*i.e.*$llc\_ref \leq llc\_ref\_thr$), it is assumed that it has more cache than it needs and label it to *Donor* and only maintains the minimum cache size (one way for Intel x86 [4]). If it has high utilization of LLC but the LLC miss rate is small, it is still a *Donor* but the cache size can only be reduced by one way in each interval until the LLC miss rate becomes non-trivial (hence labeled as *Keeper*). If this workload has significant LLC references and high LLC cache misses (*i.e.*$llc\_ref > llc\_ref\_thr$ and $llc\_miss\_rate > llc\_miss\_rate\_thr$),

[4]Intel x86 does not allow to allocate 0 way.

it may benefit from having a larger cache and is marked to *Unknown*. An *Unknown* workload that shows performance gains when getting a larger cache size is categorized as *Receiver* otherwise it is still marked as *Unknown*. If all the available cache size is used or the cache size passes a threshold (*e.g.*, 3 times the predefined cache size) and the state is still *Unknown* the workload is marked *Streaming*. *Streaming* never reuses data in cache, so it is a special *Donor* and only need to maintain the minimum cache size. In the *Receiver* state, if the workload's LLC miss rate after increasing the cache size is small or there is no further performance improvement (*i.e.* $llc\_miss\_rate < llc\_miss\_rate\_thr$ or $ipc\_imp < ipc\_imp\_thr$), dCat stops increasing the cache size and set the state to *Keeper*. At any time, *Reclaim* is applied immediately once there is a phase change detected. Since we guarantee the baseline performance for all workloads, *Reclaim* has the highest priority. If there is no available cache in the resource pool to recover the baseline cache size of *Reclaim* workloads, dCat has to reclaims cache from those whose current cache size is larger than their baseline size.



(a) Non-streaming workload.



(b) Streaming workload.

Figure 7: Example of cache allocation with dCat.

3.5 Allocate Cache Using CAT

After categorizing the workloads, the available cache needs to be distributed. In order to describe our allocation method clearly, let us use a concrete example to illustrate. Assume one tenant rents a VM (*e.g.* AWS M4.large instance) from a cloud provider to run some workloads. The cache allocation over time of this VM is shown in Figure 7(a). After getting the initial computing resource, there is no workload running at the very beginning leaving LLC underutilized. This VM thus is categorized as *Donor* and only has the minimal cache size allocated (one way). The rest of the cache are assigned to the resource pool. At time *t1*, the user starts running a memory-intensive workload (*e.g.* MLR) resulting in a phase change, and dCat immediately assigns the reserved cache size. In this particular case, the reserved cache can not fit the working set thus the LLC cache miss rate is above the threshold. If there is available cache in the resource pool, the workload cache size is increased (one way) for each round until *t2* when the LLC miss rate is smaller than the threshold and the IPC is not improved any more. At *t3*, the workload stops running, so it becomes a *Donor* again due to the low number of LLC accesses.

While, if this was a streaming workload (as in Figure 7(b)), dCat stops increasing the cache size at *t2* when the cache size reaches the *Streaming threshold* but without getting performance improvement (IPC does not change). It becomes a *Donor* after *t2*.

We notice that because dCat adds more cache to workloads lacking cache in each round, it takes some time to get the preferred state in which workloads suffers minor cache misses. This process is inevitable for the first running of each workload because dCat has no idea about the workloads' needs in advance. However, since it starts with the reserved LLC (when cache-sensitive workloads start running) it does not reduce performance from what is the expected baseline.

Table 1: Performance table for a workload phase.

Cache-ways	Normalized IPC	Mark
1	N/A	
2	0.9	
3	1.0	baseline
4	1.15	
5	1.25	
6	1.3	preferred
7	1.3	
8	1.3	

dCat maintains a table, performance table, that contains normalized IPC (normalized to baseline) collected per workload phase and cache size. This table is used to determine if a phase change has occurred, detect a sizable change on the number of memory accesses per instruction, and also to optimize the workload classification and cache allocation. This table is not necessarily complete, having all the entries filled, because only the reachable entries and/or the ones that impacts decisions are needed. For example, Table 1 shows partial table of a workload. It indicates that assigning 6 ways is enough to get the highest performance. When the same phase of this workload is seen again dCat will reuse the collected performance data for this phase skipping the process of discovery. Take the case shown in Figure 7(a) for instance, at *t4* the same phase of workload is seen again, dCat directly gives it the best number of ways instead of going to the baseline first then adding one way per round. This can reduce the time of getting enough cache to fit the working set thus further improving the overall performance. The contents of one table (mappings from number of ways to normalized IPC) are only meaningful to a specific phase, so they have to be updated once the phase changes.

This table is also essential in determining how to distribute the cache. If multiple workloads benefit from larger caches, the decision of how much and to whom to give additional cache can lead to different results. dCat supports two different allocation policies depending on which objective to achieve: 1) maximize the total performance or 2) maximize the number of workloads that achieve better performance. Maximizing total performance can lead to a single workload holding all the additional cache available (*i.e.* maximize the overall performance), in the other case every workload benefits (*i.e.*maximize the fairness) even though the best total performance may not be achieved.

To maximize the fairness among workloads, the table contents are ignored during the cache allocation. dCat evenly distributes the available cache size regardless of the magnitude of IPC improvement. For example, 2 cache-sensitive workloads run in a host while 4 cache ways are still available in the resource pool after both workloads get the baseline cache size assigned. dCat under *max fairness* mode will assign additional 2 cache ways to each regardless of their performance improvement contributed by the larger cache. Nevertheless, dCat differentiates the allocating process for *Unknown* workloads and *Receiver* workloads. Our goal is to figure out if the *Unknown* workloads are streaming or not sooner. So dCat gives *Unknown* workloads higher priority than *Receiver* workloads during cache allocation. Once *Unknown* workloads are recognized to *Streaming*, their cache-ways are transferred to those *Receiver* workloads or put into the resource pool.

To maximize the overall performance, dCat refers the performance table if table entries are valid. dCat traverses the performance tables of all workloads to find the combination whose sum of the normalized IPC of all workloads are the largest. That is for n workloads and m ways, we need find the solution from the performance table having $Max(\sum_{i=1}^{n} norm\_IPC_i)$ while $\sum_{i=1}^{n} ways_i \leq m$. This usually happens when dCat needs to re-allocate the cache size. For example, there are two workloads (*e.g.* A and B) running and both can benefit from a larger cache. Their baseline cache size is 2 ways. The CPU socket has 10 cache ways totally. At the beginning, the performance table is empty, dCat uses even distribution to assign ways, and the normalized IPC for each assignment is stored. When there are no ways available in the resource pool, each workload gets 5 ways. Their performance table contents (number of ways : normalized IPC) are (1 : N/A), (2 : 1), (3 : 1.05), (4: 1.08), (5 : 1.12) for A and (1 : N/A), (2 : 1), (3 : 1.1), (4 : 1.2), (5 : 1.25) for B. At a certain time, another workload C starts running and *reclaims* 2 ways. Since there is no way available in the pool, dCat has to collect 2 ways from A and B, then assign them to workload C to assure its baseline performance considering *Reclaim* has the highest priority. Since C's performance table is empty initially, it retains the baseline size and generates normalized IPC 1. To get the largest overall performance for A, B and C, we just need to find a combination generating the largest sum of normalized IPC for A and B. After traversing A's and B's performance table, we realize giving A 3 ways and B 5 ways is the best choice, because this combination generates the largest total normalized IPC (*i.e.* 2.3 for A plus B, and 3.3 for A, B and C).

4 IMPLEMENTATION

We implemented a prototype of dCat for Intel Xeon (Xeon-D and Xeon-E5 v4) platform. dCat is a C program and runs as a daemon in the host OS (Linux 4.4 is used for our prototype). dCat daemon periodically performs the five steps demonstrated by Figure 4. It is transparent to the VMs, guest OS and all user-level applications.

Workload Profiling To determine the phase change and categorize the workloads, we have to record and analyze a workload's behavioral characteristics. This is done by profiling CPUs where the workload is running, that is, collecting CPU metrics like LLC cache miss and L1 cache reference. We use a Linux kernel module named *msr* to read a series of performance events from processor counters[5]. Some important performance events used by dCat are listed in Table 2. The IPC is the ratio of retired instructions to unhalted cycles.

Table 2: Performance events used by dCat.

	Event Num.	Umask Value
LLC Misses	2EH	41H
LLC References	2EH	4FH
L1 Cache Misses	D1H	08H
L1 Cache Hits	D1H	01H
Retired Instructions	309	
Unhalted Cycles	30A	

dCat determines a workload phase by memory accesses (equal to LOAD + STORE) per instruction. However, *msr* is unable to count the issued LOAD and STORE instructions. Since x86 CPU generally checks the fastest, L1 cache first before consuming a data, we use L1 references value to estimate the memory accesses number. One workload phase, accordingly, is defined by $l1\_ref/ret\_ins$.

Cache Allocation dCat leverages the Intel Platform Quality of Service (pqos) library [21] (which provides the interface to CAT) to dynamically set the mapping between the cache size and CPU cores. This library supports up to 16 Class of Service (COS). Each COS represents a subset of LLC. dCat dynamically defines the COSs according to the cache requirement of each workload and applies them to the corresponding cores where workloads are running. Since existing Intel commodity processors only allow the way level partitioning, dCat assigns cache in the unit of cache-way. To guarantee the cache isolation, we do not allow the COS overlap among cores, that is the cores running one workload (or a set of workloads owned by the same tenant) only have one COS.

We periodically collect the CPU/cache information and change the cache allocation for different workloads owned by different tenants. The period time is a configurable parameter (*e.g.* 1s). It can be changed by the administrators according to their needs. Note that if the interval is too large, dCat can not adjust the cache allocation on demand. If the value is too small resulting in very frequent change of cache assignment, it would introduce non-trivial overhead. Besides, we use IPC as the feedback of cache adjustment, the benefit of extra cache may not be reflected within the very short interval.

Since LLC can only be assigned to CPU cores, dCat needs the information that where the workloads (VMs or containers) are running. In bare-metal, it can be collected from OS which maintains the footprints of all applications. In public or private cloud, containers and VMs are widely used to consolidate the physical servers. We can get this information from the hypervisor which schedules vCPUs on physical cores. To be simplified, we assume there is no CPU over provisioning in our prototype, that is no CPU sharing among vCPUs. So, each VM/container has dedicated CPU resource.

[5]Intel Cache Monitor Technology (CMT) misses some metrics such as L1 reference, phase change detection etc. that we need for dCat. CMT only reports statistics but cannot integrate with CAT to dynamically allocate cache.

5 EVALUATION

This section presents our evaluation of dCat using both micro-benchmarks and some real world applications.

Evaluation Setup Our experimental system consists of a server with Intel Xeon E5-2697 v4 processor with 18 cores at 2.3 GHz and a 20-way 45 MB LLC. The capacity of each cache way is 2.25 MB. The host server runs KVM as the hypervisor and Linux 4.4 as the OS for both the guest VM and the host. To avoid the impact of CPU frequency scaling, all CPUs' frequency governors are set to *performance*. For Redis, PostgreSQL and Elasticsearch, the server is connected to a client using a 10 Gbps network.

To simulate the multi-tenant cloud environment, all benchmarks and background workloads are running in VMs representing independent tenants. All VMs in our experiments are assigned 2 vCPUs and 4GB RAM each. We pin each vCPU to separate physical threads in the host, giving every VM dedicated CPU resources. KVM vhost-net is enabled to boost the virtual network performance. All performance results *e.g.*data access latency, running time, bandwidth etc. shown in this section are obtained from the application side.

5.1 Microbenchmark Results

Parameters Sensitivity dCat have a series of parameters to decide if the corresponding operations occur or not. The most import two are cache miss parameter (*llc_miss_rate_thr*) and IPC improvement parameter (*ipc_imp_thr*). The former one decides how sensitive of dCat to cache miss. Once the cache miss rate of a VM is larger than this threshold, dCat assumes one or more cache sensitive workloads are running and try to assign more cache to fit their working set. The later one affects how dCat categorizes workloads (*i.e.* if *Receiver* or not). If one's IPC improvement is larger than *ipc_imp_thr* after getting additional cache, it becomes *Receiver* and may continue getting more cache if it is available.

Figure 8 displays the relationship between cache allocation, data access latency and the value of cache miss threshold when a MLR with 8MB working set run in the target VM. VM's baseline cache size is set to 2 ways. The results are collected after running MLR for 30s when the cache-ways assignment does not change (*i.e.*getting the *preferred* cache ways). This experiment indicates smaller cache miss parameter predicts the cache requirement more accurately and brings better performance. Nevertheless, small value leads to higher cache demand and the cache pool drains sooner. Cloud providers can set this threshold to their desired values according to their hardware models and customers' needs. We choose 3% in our following experiments.

Figure 9 presents the cache allocation when the target VM retains the role of *Receiver*. The workload and experimental setup are same to the cache miss threshold experiment above. We vary the IPC improvement threshold from 3% to 40%. The 9 cache ways for 3% threshold means the VM achieves more than 3% IPC improvement for every additional cache way assigned until having 9 cache ways. Like cache miss threshold, smaller IPC improvement represents higher sensitivity and accuracy, but it also means higher pressure on the cache resource. We set this threshold to 5% in our following experiments.

Cache Allocation To verify that dCat can dynamically adjust the number of ways according to the workload's behavior, we repeat

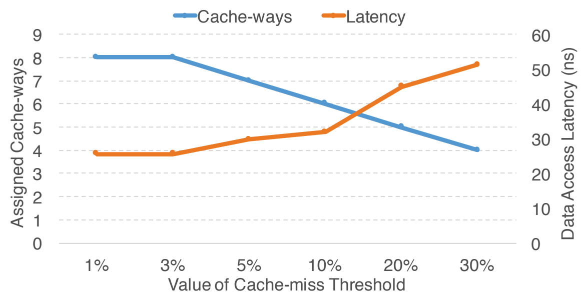


Figure 8: Impact of cache miss threshold

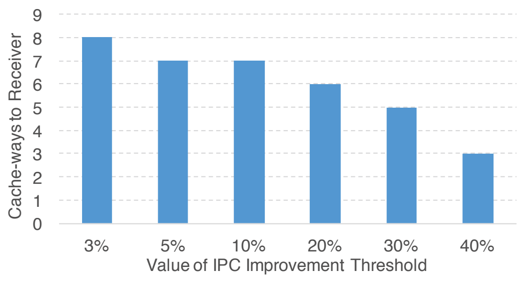


Figure 9: Impact of IPC improvement threshold

some similar experiments shown in Section 2 that run a workload with a larger working set than the allocated LLC size. In Section 2 we showed that static cache partitioning via CAT works poorly for this scenario due to lacking cache capacity and associativity. Here, 6 VMs each with a baseline of 6.75MB LLC (3 ways) are running in the host machine. We run MLR in a VM (*i.e.* target VM) and track the cache allocation of this VM under the management of dCat. The working set of MLR is varied from 4MB to 16MB. We run lookbusy [2] which consumes CPU but does not have significant cache access in the rest 5 VMs. This experiment shows that dCat can dynamically assign the correct amount of cache to the VM hosting MLR (shown in Figure 10). In this case, lookbusy does not benefit from increased cache so that each lookbusy VM only holds 1 cache way (*classified as Donor*) and all rest cache ways on the socket will be gradually assigned to the MLR VM until its performance in terms of IPC stops increasing.

The overhead of dCat is measured. We observed that the CPU utilization of dCat is always below 1% when the above experiment is running, which means the CPU overhead introduced by dCat is negligible.

We also measure the data access latency when running the MLR experiments above. Figure 11 shows the normalized latency over the full cache case in which MLR occupies all cache-ways of the CPU socket which is far enough to hold its whole working set. We can see that dCat's normalized latency is just slightly higher than the full cache scenario and much better than the static partition with CAT.

What Figure 10 shows is the cache allocation and normalized IPC (to baseline) for workloads' first running. Since no performance history can be referred to, the cache-way is incremented by one each round (except the *reclaim*). If the period of each round is one

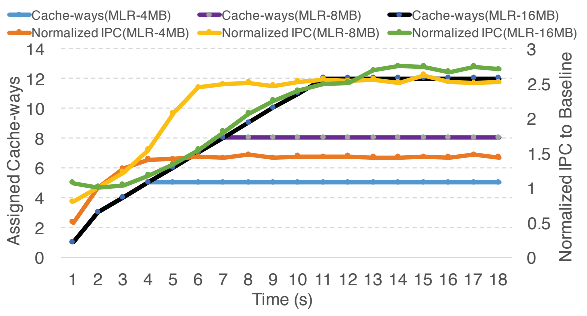


Figure 10: Cache-way allocation and normalized IPC (to baseline) for MLR

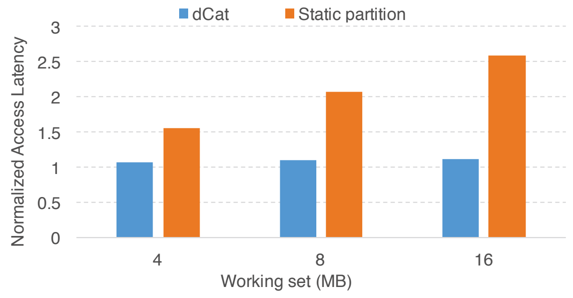


Figure 11: Normalized (to *full cache*) data access latency for MLR

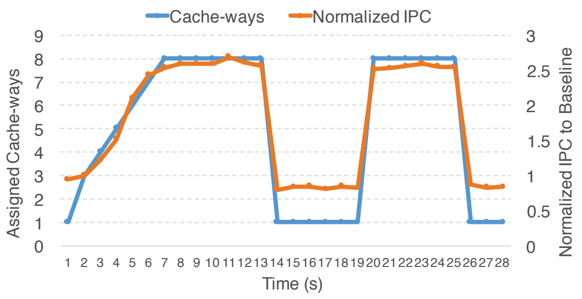


Figure 12: Cache-way allocation and normalized IPC (to baseline) for MLR with repeated running

second (configured to 1s in evaluation section), it takes up to 7 seconds (for MLR-8MB) or up to 11 seconds (for MLR-16MB) to get enough cache for needs, because dCat does not know how many cache needed in advance and have to try different number. While, when the same workloads stop then run again, this process can be accelerated with the help of performance table introduced in Section 3.5. Figure 12 shows this scenario. We run the MLR-8MB in the VM. The first running ends at 14s. At 19s the MLR-8MB starts running again and directly gets 8 cache-ways. The second running ends at 26s. The results shown in this figure matches our expectation displayed by Figure 7(a) in Section 3.5 very well.

Figure 13 shows the cache allocation and normalized IPC of running MLOAD (with 60MB working set) instead of MLR in the target

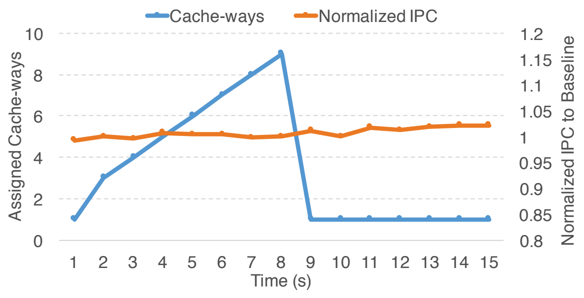


Figure 13: Cache-way allocation and normalized IPC (to baseline) for MLOAD

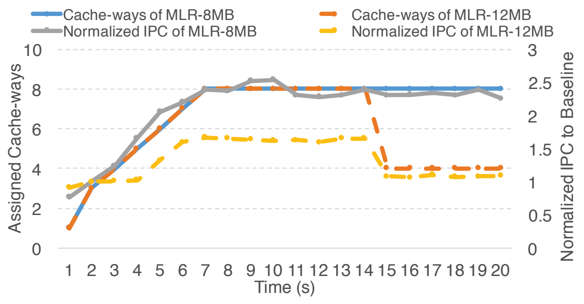


Figure 14: Cache-way allocation and normalized IPC (to baseline) for 2 MLR

VM. Since 60MB is much larger than the available cache on the socket, MLOAD-60MB generates cyclic data access pattern [35] so that can not benefit from additional ways. Its IPC does not change despite increasing cache size. When its cache allocation reaches the streaming threshold (*e.g.* 3 times of the baseline allocation), it is classified as *streaming* and its allocation is changed to only 1 way. The results closely match our expectation displayed by Figure 7(b) in Section 3.5. The cache taken away from streaming workloads are put into a free pool and can be assigned to other workloads that would benefit from more cache. If static cache partitioning was used, this workload would always consume its initial allocation (3 cache-ways) even if is not useful, wasting 2 ways. If the cache was shared in an uncontrolled way, the high memory pressure from MLOAD-60MB would slow down other workloads by evicting their data.

Multiple memory-intensive workloads (or VMs) may run on the same host machine so we evaluate the efficiency of dCat for two simultaneous memory intensive VMs. First, we run 2 MLR (one with 8MB working set, the other with 12MB) in two VMs while 4 other VMs run lookbusy. Let us recall that in Section 3 we describe two cache allocation policies supported by dCat. One is the even distribution (better fairness), the other one is allocating based on performance table (maximized overall performance). With the *fair policy*, each VM get the same cache allocation if there are ways available in the free pool. We demonstrate the performance-maximizing policy in Figure 14. During the first 14 seconds the two policies behave the same because the performance table is empty and there are enough ways available in the free pool. The two VMs get equal cache size each step. When each one gets 8 ways, there is no more

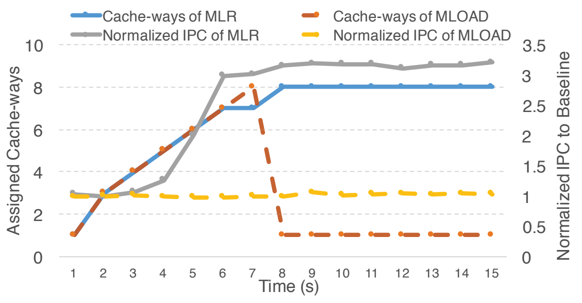


Figure 15: Cache-way allocation and normalized IPC (to baseline) for MLR and MLOAD

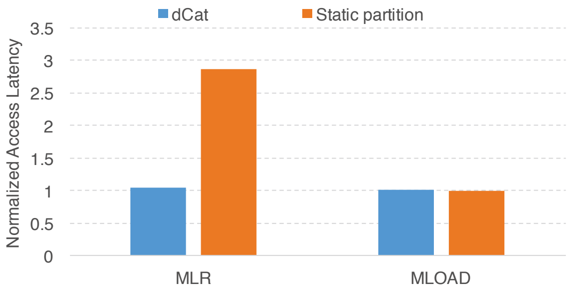


Figure 16: Normalized (to *full cache*) data access latency with dCat for MLR and MLOAD

cache in the pool (the 4 background VMs collectively use the other 4 ways). At 14 s, we start another 4 VMs with one way each on the same socket resulting in a cache reallocation. At this time, only 12 ways instead of 16 can be assigned to two MLR VMs. Based on the recorded normalized IPC in the performance table, the combination of 8 ways to MLR-8MB and 4 ways to MLR-12MB achieves the highest aggregate normalized IPC. The reason behind is that, considering the random data access pattern of MLR, for certain amount of cache MLR-8MB has higher probability to read the data cached in LLC than MLR-12MB. Hence, giving ways to MLR-8MB results in higher normalized IPC.

We also try the scenario of running one MLR with 8 MB working set and one MLOAD with 60 MB working set (*noisy neighbor*). We add one more VM in this experiment than above; the other 5 VMs run lookbusy. The results are shown in Figure 15 and Figure 16. Figure 15 shows the cache allocation and normalized IPC over time. They both *reclaim* the baseline cache (3 ways) first then get one additional each round until 6s when each one obtains 7 cache ways. At 7s, there is only one cache-way available (the 5 background VMs use the rest 5 cache-ways). Since *Unknown* (current categorization of MLOAD-60MB) has higher priority than *Receiver* (current categorization of MLR-8MB), MLOAD-60MB gets the last cache-way. Then MLOAD-60MB releases 7 cache ways at 8s because it has not IPC improvement when all available cache are consumed (classified as *Donor*), leading to more cache ways available in the free pool. MLR-8MB stops receiving more cache way after getting one of the released cache ways by MLOAD-60MB. MLR-8MB's final cache allocation is 8 ways which is its preferred state. From Figure 16 we can see that when the MLR's performance is improved by around 175% via dCat, the performance of MLOAD does not get hurt. It means we fully utilize the unused cache ways in the system to improve the performance of those workloads which indeed need the cache.

5.2 Benchmark/Application Results

This section shows the results of some more complex benchmarks and real world applications with dCat.

First, we use the SPEC2006 [19] benchmark suite to measure the performance of dCat. In this experiment, 5 VMs with a baseline of 4 cache-ways (9MB total) each are running in the system. We run the selected (single-threaded) SPEC2006 benchmarks in one VM and run 2 MLOAD (60 MB working set) in two separate VMs as the noisy neighbors. The rest two VMs run lookbusy, acting as polite neighbors. The SPEC2006 benchmark outputs the running time of each workload, the results in Figure 17 are the reciprocal value of the running time and are normalized to the results with shared cache. Since the different SPEC2006 benchmarks have different working set size and different sensitivity to the cache [24], cache isolation and allocation have different effects for the subtests. Some workloads get better performance with dCat but some do not, but at least achieve the same performance compared with static partition and shared cache. For most SPEC benmarks, static partitioning via CAT achieves the same or better performance compared with the shared cache scenario because it prevents warm data from being evicted by noisy neighbors (MLOAD-60MB in this case). However, when the working set size is larger than the reserved cache partition, the resulting performance is far from optimal. dCat provides better performance than CAT due to the extra cache collected from other VMs which do not benefit from larger cache. For different workloads, the benefit from dCat is different. Take *omnetpp* and *astar* for example—the ratio of core working set size (CWSS) to working set size (WSS) of these two workloads is high [16]. It means they have high reuse of the data in the working set, so the larger cache capacity granted by dCat gives more benefit (up to 83% compared with static cache and up to 129% over the shared cache). The geo-mean performance of 20 chosen benchmarks is improved by 25% over the shared cache and by 15.7% over static partitioning. The assigned ways by dCat for each benchmark (or different data set) are shown in Table 3. Each benchmark runs hundreds of seconds so the way assignment changes during the run. The values listed in the third column are the ceiling number during the run. The cache allocation changes are caused by either the workload characteristic or the phase changes.

Next, we show the results of three cloud applications. The experimental setup is similar to the SPEC2006 experiments. We run the tested application in one out of five VMs. Two VMs run MLOAD-60MB as the memory-intensive noisy neighbors. The other two VMs run lookbusy.

We first evaluate the performance of the Redis [44] in-memory key/value store. Since Redis keeps all datae in memory, cache is critical to performance. We run the Redis server in the target VM and run the client side benchmark Memtier [3] in another host within the same LAN. 1 million records (128 bytes each) are inserted to Redis server first as the sample data. Then Memtier generates many concurrent *Get* requests (using 8 threads and pipeline depth of 30)

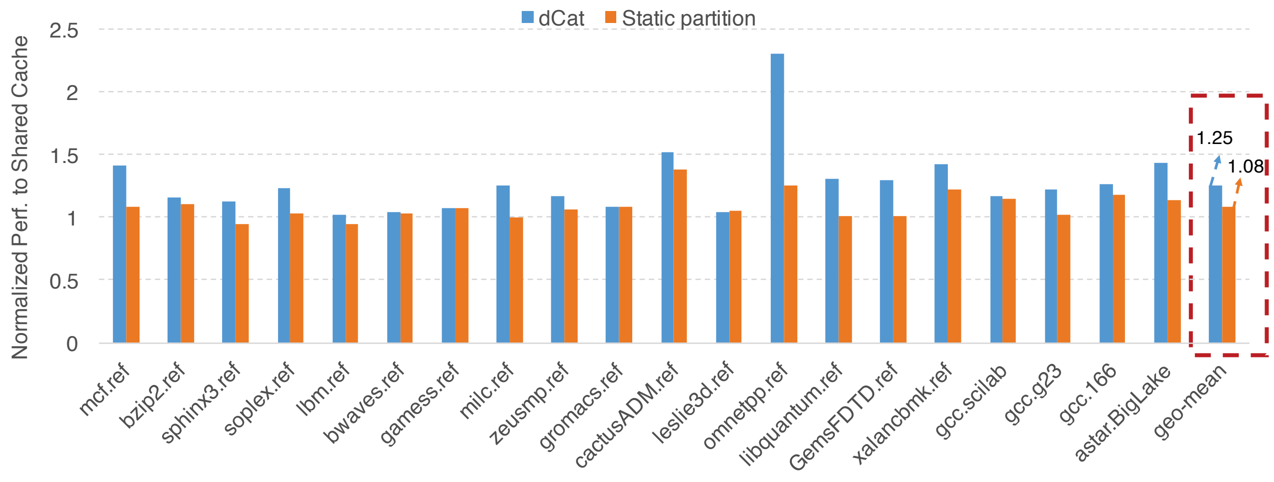


Figure 17: Normalized performance to *shared cache* of SPEC2006 with adversary workloads

Table 3: Characteristic and assigned ways of SPEC2006.

Benchmark	Input	Assigned Cache-ways	Characteristic
mcf	ref	up to 11	both
bzip2	ref	up to 7	interference dominated
sphinx3	ref	up to 9	capacity dominated
soplex	ref	up to 9	both
lbm	ref	up to 8	capacity dominated
bwaves	ref	up to 9	both
gamess	ref	up to 5	capacity dominated
milc	ref	up to 10	capacity dominated
zeusmp	ref	up to 9	capacity dominated
gromacs	ref	up to 5	interference dominated
cactusADM	ref	up to 9	capacity dominated
leslie3d	ref	up to 5	both
omnetpp	ref	up to 9	capacity dominated
libquantum	ref	up to 9	capacity dominated
GemsFDTD	ref	up to 9	both
xalancbmk	ref	up to 8	both
gcc	166	up to 5	both
gcc	g23	up to 8	both
gcc	scilab	up to 7	interference dominated
astar	biglake2048	up to 8	both

Table 4: Performance improvement for Redis with dCat.

	IOPS (operations/s)	Latency (ms)	Assigned Cache-ways
Shared	684333	18.657	N/A
Static Partition	851797	14.327	4
dCat	1078558	12.195	up to 7

to the Redis server. The measured IOPS and latency of reading data from Redis are shown in Table 4. In this experiment, dCat gives the Redis server up to 3 additional ways compared to static allocation and thus improves IOPS by up to 26.6% over the static partition and up to 57.6% over the shared cache.

Table 5: Performance improvement for PostgreSQL with dCat.

	TPS (transactions/s)	Latency (ms)	Assigned Cache-ways
Shared	13980.59	0.281	N/A
Static Partition	13171.45	0.297	4
dCat	14802.36	0.265	up to 6

PostgreSQL [4] is a widely used open source object-relational database system. It caches table data, indexes, and query execution plans in a LRU based memory buffer. A larger cache is helpful for reading data faster from the database. We run the PostgreSQL server in the target VM and run pgbench [5] benchmark issuing *select* queries from another host in the same LAN. 10 million tuples are inserted into PostgreSQL database initially. Table 5 displays the measured TPS (transactions per second) including connections establishing and average latency per *select* transaction. With the assigned additional cache-ways, dCat achieves 10.7% lower latency than static partition and performs around 5.7% better than the shared cache. We also tried the multiple database instances scenario in which 3 PostgreSQL instances run in 3 separate VMs (the adversary workloads are still MLOAD-60MB and lookbusy), we observed the similar improvement with dCat.

Table 6: Performance improvement for Elasticsearch with dCat.

	Average Latency (us)	95 Percentile Latency (us)	99 Percentile Latency (us)	Assigned Cache-ways
Shared	75.5	94	182	N/A
Static Partition	76.72	92	181	4
dCat	69.2	86	160	up to 9

Elasticsearch [1] is a widely-used search and analytics engine. We use the YCSB [11] cloud benchmark suite to measure the performance of Elasticsearch. We use workload C which reads from a database of 100K records, each 1 KB. The results are shown in Table 6; dCat improves average latency by 10% and 99th percentile latency by 11.6% over both static partitioning and shared cache.

6 DISCUSSION

Since dCat is a software solution based on Intel CAT, it inevitably inherits some limitations/restrictions:

- The cache allocation knob resides on each CPU core (or thread if hyper-threading is enabled). In order to guarantee

the performance isolation among tenants, the VMs/containers have to own dedicated CPU cores/threads.
- Intel Xeon processors currently support up to 16 COS, thus the isolated VMs/containers per socket can not exceed 16. Meanwhile, the VM/container number should be also smaller than the number of ways of cache associativity.
- Intel does not have an instruction to clear a specific cache-way. To explicitly clear a cache-way after changing the cache allocation, one user-level cache flush application (*e.g.* reading an array sequentially) is necessary.

7 RELATED WORK

We have introduced some alternative solutions for LLC partitioning in Section 2. In this section, we discuss other related work in the same research area.

Performation isolation in memory hiearchy Subramanian et al. [39] developed an Application Slowdown Model that accurately estimates the slowdown caused by cache and memory interference, also using hardware extensions. [31] presents a characterization methodology that enables the accurate prediction of the performance degradation that results from contention for shared resources in the memory subsystem. However, they did not propose to mitigate the performance interference caused by the shared LLC which dCat targets.

CPI2 [45] is a technique for semi-black-box performance isolation that compares the IPC of multiple running copies of a workload and looks for outliers to find performance interference, then throttles co-scheduled workloads to reduce interference. Similarly, Quasar [14] is a cluster scheduler that profiles workloads and uses that information to predict allocation sizes and placements for future runs that will maximize performance and minimize interference. Instead of hardware or kernel controls, it relies on workload parameters such as memory and number of threads to control resource usage. Both of these techniques rely on profiling multiple copies of each workload, while dCat is designed to be effective in environments where such knowledge is not available.

Heracles [30] is one of the first systems to use Intel CAT and it uses two levels of closed-loop control to isolate latency-critical workloads from interference caused by background best-effort jobs. It can dramatically increase the utilization of a cluster by safely running much more best-effort work without violating service level objectives. Heracles is not a general performance isolation system, though, since it assumes only two workload classes and it requires the latency-critical workload to provide performance metrics. It mainly targets some specific workloads in google's data centers. In a public cloud each server can host more than two workloads and the workloads may be black boxes that provide no performance feedback. dCat does not rely on any feedback from user-level applications, and it supports multiple workloads from different tenants.

Ginseng [15] proposes auction-based dynamic cache allocation using Intel CAT for black-box workloads in a public cloud. Instead of optimizing metrics like IPC, throughput, or latency, Ginseng attempts to maximize the profit produced by the cache. Using auctions allows each workload to have independent and arbitrary utility functions, but it also shifts the burden of workload performance analysis to the cloud customer and creates bin-packing problems since cache demand may not be proportional to other resources on the machine.

Page coloring [28, 43] offers a different cache partitioning approach which is orthogonal to dCat. It allocates different cache sets to different workloads by controlling the physical address assignment for those workloads' pages. Unfortunately, when LLC allocation changes, this approach has to re-assign physical address to some pages and thus induces the overhead of copying those pages between different memory locations. Due to its complexity and non-trivial overhead, Linux does not support this method in kernel so far.

Performation isolation in I/O Performance interference and isolation in I/O subsystem has been studied extensively. SecondNet [18] and Oktopus [7] statically allocate the network bandwidth to guarantee the bandwidth reservation but may lead to the underutilized network (not work-conserving). Elasticswitch [34], EyeQ [25] and SliverLine [33] all provide the network isolation with the work-conserving model. However, all of them above did not consider the CPU interference caused by shared LLC which is targeted by this work.

In order to provide storage isolation among VMs, mClock [17] modifies hypervisor's I/O scheduler to differentiate VMs' I/O limits and reservations. Argon [40] uses cache management and time-sliced disk scheduling to guarantee the performance isolation in a shared storage server. Stout [32] utilizes batch processing to minimize storage request latency, but does not consider the fairness among tenants.

8 CONCLUSION

The shared last level cache (LLC) in Intel's latest x86 architecture provides an effective design for maximizing the utilization of cache in a processor socket. However, when multiple workloads run on a processor, the interference in the cache can lead to poor/inconsistent performance. This can be detrimental to operations in a public-cloud environment or any where predictable performance is needed. Intel's CAT technology gives the means to provide cache isolation by limiting the number of ways mapped to a particular core at any given instant. Static application of it is enough to provide consistent performance, but it can cause under-utilization of cache and poorer performance per workload than a shared cache when the static partitioning does not quite match the individual needs for the workloads.

In this work we design, implement and evaluate a dynamic cache allocation scheme, dCat. dCat leverages CAT for setting cache allocations (partitions) to multiple concurrently executing workloads on a socket and combines it with continuous assessment and dynamic re-sizing of allocations based on the real-time needs of the workloads. This allows dCat to get both a consistent minimum bound on performance for each workload irrespective of its neighbors sharing the socket as well as improve workload performance when the neighbors are not making beneficial use of their allocations. We explain the issues and analyze the fundamental workings of dCat with micro-benchmarks and evaluate them with SPEC CPU2006, Redis, PostgreSQL and ElasticSearch as workloads. dCat gets consistently the best performance for a workload compared to fully shared cache or static application of CAT while preserving the performance isolation possible with CAT.

REFERENCES

[1] Elasticearch. https://https://www.elastic.co/.

[2] lookbusy – load generator. http://www.devin.com/lookbusy/.

[3] Memtier. https://github.com/RedisLabs/memtier_benchmark.

[4] The PostgreSQL. https://www.postgresql.org.

[5] The PostgreSQL Benchmark. https://www.postgresql.org/docs/devel/static/pgbench.html.

[6] Gartner says worldwide public cloud services market to grow 18 percent in 2017. Gartner Press Release: https://www.gartner.com/newsroom/id/3616417, Feb 2017.

[7] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 242–253.

[8] BIRCHER, W. L., AND JOHN, L. K. Core-level activity prediction for multicore power management. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems 1*, 3 (2011), 218–227.

[9] CHETSA, G. L. T., LEFEVRE, L., PIERSON, J.-M., STOLF, P., AND DA COSTA, G. A user friendly phase detection methodology for hpc systems' analysis. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing* (2013), IEEE, pp. 118–125.

[10] COLLINS, J. D., AND TULLSEN, D. M. Hardware identification of cache conflict misses. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture* (1999), IEEE Computer Society, pp. 126–135.

[11] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[12] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[13] DELIMITROU, C., AND KOZYRAKIS, C. ibench: Quantifying interference for datacenter applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on* (Sept 2013), pp. 23–33.

[14] DELIMITROU, C., AND KOZYRAKIS, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS 14, ACM, pp. 127–144.

[15] FUNARO, L., BEN-YEHUDA, O. A., AND SCHUSTER, A. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016).

[16] GOVE, D. Cpu2006 working set size. *ACM SIGARCH Computer Architecture News 35*, 1 (2007), 90–96.

[17] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), USENIX Association, pp. 437–450.

[18] GUO, C., LU, G., WANG, H. J., YANG, S., KONG, C., SUN, P., WU, W., AND ZHANG, Y. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference on Emerging Networking EXperiments and Technologies* (2010), ACM, p. 15.

[19] HENNING, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[20] HERDRICH, A., VERPLANKE, E., AUTEE, P., ILLIKKAL, R., GIANOS, C., SINGHAL, R., AND IYER, R. Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2016), IEEE, pp. 657–668.

[21] INTEL. Increasing platform determinism with platform quality of service for the data plane development kit. *Intel White Paper 2* (2016).

[22] IYER, R. Cqos: A framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th Annual International Conference on Supercomputing* (New York, NY, USA, 2004), ICS '04, ACM, pp. 257–266.

[23] IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), SIGMETRICS '07, ACM, pp. 25–36.

[24] JALEEL, A. Memory characterization of workloads using instrumentation-driven simulation–a pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites. *Intel Corporation, VSSAD* (2007).

[25] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., KIM, C., AND AZURE, W. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *HotCloud* (2012).

[26] KAMBADUR, M., MOSELEY, T., HANK, R., AND KIM, M. A. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 51:1–51:12.

[27] KASTURE, H., AND SANCHEZ, D. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS 14, pp. 729–742.

[28] LIEDTKE, J., HARTIG, H., AND HOHMUTH, M. Os-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE* (1997), IEEE, pp. 213–224.

[29] LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on* (2008), IEEE, pp. 367–378.

[30] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (New York, NY, USA, 2015), ISCA 15, ACM, pp. 450–462.

[31] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture* (2011), ACM, pp. 248–259.

[32] MCCULLOUGH, J. C., DUNAGAN, J., WOLMAN, A., AND SNOEREN, A. C. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference* (2010), pp. 47–60.

[33] MUNDADA, Y., RAMACHANDRAN, A., AND FEAMSTER, N. Silverline: Data and network isolation for cloud services. In *HotCloud* (2011).

[34] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., TURNER, Y., AND SANTOS, J. R. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication* (2013), 351–362.

[35] QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News* (2007), vol. 35.2, ACM, pp. 381–391.

[36] QURESHI, M. K., AND PATT, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture, 2006. MICRO-39.* (2006), IEEE, pp. 423–432.

[37] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011), ISCA 11, ACM, pp. 57–68.

[38] SHERWOOD, T., SAIR, S., AND CALDER, B. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News* (2003), vol. 31.2, ACM, pp. 336–349.

[39] SUBRAMANIAN, L., SESHADRI, V., GHOSH, A., KHAN, S., AND MUTLU, O. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture* (2015), MICRO-48, ACM, pp. 62–75.

[40] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *FAST* (2007), vol. 7, pp. 5–5.

[41] XIE, Y., AND LOH, G. Dynamic classification of program memory behaviors in cmps. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects.* (2008).

[42] XIE, Y., AND LOH, G. H. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News* (2009), pp. 174–183.

[43] YE, Y., WEST, R., CHENG, Z., AND LI, Y. Coloris: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), ACM, pp. 381–392.

[44] ZAWODNY, J. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine 79* (2009).

[45] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys 13, ACM, pp. 379–391.