CoPlace: Effectively Mitigating Cache Conflicts in Modern Clouds

Xiaowei Shang[1], Weiwei Jia[2], Xiaoning Ding[3]
Department of Computer Science
New Jersey Institute of Technology
Newark, NJ, USA
{[1]*xs225,* [2]*wj47,* [3]*xiaoning.ding*}*@njit.edu*

Jianchen Shan
Department of Computer Science
Hofstra University
Hempstead, NY, USA
Jianchen.Shan@hofstra.edu

***Abstract*—Substantial renovations in hardware cache have been focused on reducing cache interference between workloads recently. However, cache conflicts within each workload are surprisingly overlooked. The paper identifies that cache conflicts cannot be effectively reduced in virtualized clouds. Enhancements for cache partitioning, such as Intel cache allocation technology, make cache conflicts even more serious for cloud workloads.**

The paper proposes CoPlace as a low overhead and highly portable solution for virtualized clouds. CoPlace enhances the page placement mechanisms implemented in the host OS, such that it can collaborate with the guest OS to reduce cache conflicts. With CoPlace, the guest OS makes page placement decisions; and the host OS helps enforce the decisions.

Evaluation based on the prototype implementation in Linux and KVM and diverse real world applications shows that CoPlace can significantly reduce cache conflicts and improve application performance.

***Keywords*-Cache Conflicts, Memory Management, Virtualization, Page Coloring, Page Placement**

I. Introduction

Targeting cloud platforms, extensive research efforts have been focused on reducing cache interference between workloads, which is caused by workloads sharing last level caches (LLCs). The solutions are LLC partitioning with various techniques [1]–[17]. Recently, hardware enhancements in processors, such as Intel CAT (cache allocation technology) and AMD CAE (cache allocation enforcement) are designed and utilized to partition LLCs [2], [3], [5], [17].

While cache interference between workloads is intensively studied, non-interference caching problems suffered by individual workloads (e.g., those occur even in dedicated cache space) are largely ignored on cloud platforms. How LLC partitioning impacts these problems particularly lacks investigation. Examining these problems becomes increasingly important on the latest cloud platforms with workloads using dedicated LLC partitions.

This paper shows that, in a virtualized cloud, workloads suffer from increased last level cache conflicts, even when there is no cache interference from other workloads. The conflicts become more serious with LLC partitioning, because LLC partitioning reduces the associativity of the LLC space available to each workload and makes it more difficult for a hardware cache to reduce cache conflicts. Even worse, on the processors with non-inclusive LLCs, where L2 cache misses incur memory accesses, workloads also suffer from increased L2 cache misses, as shown in §III.

Increased LLC conflicts are incurred in virtualized clouds, because memory page placement mechanisms (e.g., page coloring and bin hopping [1], [18]–[20]) become ineffective in reducing LLC conflicts on these platforms. Implemented in system software, page placement mechanisms reduce cache conflicts by improving the allocation of memory pages to workloads. They are important measures particularly when cache associativities are too low to effectively absorb cache conflicts. Leveraging the fixed mapping between pages (i.e., page physical addresses) and cache sets, they first identify conflicting pages (i.e., pages mapped to the same cache sets), and then allocate non-conflicting pages to hold the data to be accessed together. Different mechanisms allocate non-conflicting pages with different page placement policies. For example, page coloring assumes sequential memory accesses and allocate non-conflicting pages to the data with contiguous virtual addresses. Bin-hopping targets repetitive memory access patterns and allocates non-conflicting pages to the data consecutively accessed by a workload.

In virtualized clouds, page placement mechanisms are equipped at both guest and host layers. However, as the paper shows in §III, they cannot effectively reduce cache conflicts, no matter how different page placement mechanisms are used combinatorially at these layers. For a workload on a virtual machine (VM), the memory pages it can use are determined by two independent page allocations, which are separated by the semantic gap produced by virtualization: In the host OS, host physical pages are allocated to the VM without knowing how they will be allocated to workloads. In the guest OS, guest physical pages are allocated to the workload without knowing whether they are conflicting pages or not. Since neither the host OS nor the guest OS has a complete control over the page placement of the workload, neither page placement mechanism can effectively reduce cache conflicts. Due to the independent page allocations and the semantic gap between the host and the guest, their page placement mechanisms cannot collaborate synergistically in reducing cache conflicts.

The nature of virtualization excludes the unification of two layers of page allocations into one that directly allocates host physical pages to the workloads in VMs. Thus, to reduce cache conflicts, the paper proposes to create and maximize the synergy between the page placement mechanisms at the two layers. The synergy requires the interaction between the guest and the host. Thus, the key issue in the implementation of the idea is how to minimize the frequency and complexity of the interaction. Frequent interaction incurs high overhead because costly context switches must be involved. Complex interaction may require the extension to the guest-host interface and/or substantial changes to the guest OS, limiting the portability of the solution.

The paper presents COPLACE as a low-overhead and highly portable solution. COPLACE does not introduce extra guest-host interaction to existing systems, and does not change the guest-host interface or guest OSs. COPLACE achieves this with the following two designs.

First, COPLACE eliminates the interaction required for making page placement decisions (i.e., which data in a workload should be allocated with non-conflicting pages) by relying solely on the guest OS for decision making. The host does not help the guest in decision making to avoid cross-layer interaction. It does not make its own page placement decisions either to avoid decision conflicts. (The page placement policies implemented in the existing host OS design are removed with COPLACE.) The page placement mechanisms in the host only helps enforce the page placement decisions made by the guest, in order to achieve synergy: when the guest determines to use non-conflicting pages to hold some data, the host ensures that the corresponding host physical pages are really non-conflicting.

Second, COPLACE uses guest physical page addresses to convey page placement decisions. This eliminates the the interaction required for transferring page placement decisions to the host and avoids changing the existing guest-host interface. Specifically, with existing system designs, the host presents a virtual LLC to each virtual CPU (vCPU) socket [21]. Thus, the guest OS can perform page placement in the same way as it does on a physical machine. In the VM, to map guest physical pages to the cache sets in a virtual LLC, the guest OS uses a fixed set of bits in the guest physical addresses of the pages. Thus, page placement decisions can be inferred from guest physical page addresses: the guest physical pages with the same value on these bits are considered as conflicting pages, and should be backed by conflicting host physical pages; the pages with different values on these bits are considered to be non-conflicting, and should be backed by non-conflicting host physical pages. The host must enforce these decisions inferred from guest physical page addresses when allocating host physical pages to VMs, and maintain the decisions when performing memory ballooning or deduplication.

The paper makes the following contributions. First, to our knowledge, this is the first work that identifies and studies the cache conflict problem and its causes on modern processors in clouds. Second, we have proposed COPLACE as an effective solution that can efficiently address the problem and the technical challenges of the solution. Finally, we have implemented COPLACE based on KVM in Linux kernel 5.3 and tested it with diverse applications. Our tests show COPLACE can significantly reduce cache conflicts and effectively improve application performance and system efficiency.

II. Background

A. Reducing Cache Interference in Clouds

In clouds, with the fast growing memory capacities (e.g., Amazon EC2 offering up to 24 TB in one instance [22]), accessing the data sets that fit into memory has become a performance bottleneck for many workloads [10], [23]–[28]. To accelerate the memory accesses of cloud workloads, hardware cache architecture and system software recently are undergoing substantial renovations, focusing on reducing cache interference. Cache interference is one of the major sources that increase cache misses in clouds. It happens in a LLC shared by multiple workloads, where loading data for one workload evicts the data that is to be accessed by another workload.

A popular solution is LLC partitioning [1]–[17]. For example, Intel CAT and AMD CAE allow software to control the partitioning of the LLC (each partition being one or multiple cache ways) and the allocation of LLC partitions to different workloads [3]–[5].

A growing number of processors (e.g., Intel Scalable Processors and AMD Zen2/3 Processors) are making their LLCs non-inclusive and using increasingly large L2 caches (e.g., 1MiB per core in Intel Xeon Gold 6138). This also helps reducing cache interference, because workload performance relies less on LLC performance. With the non-inclusive cache design, the data in L2 caches may not exist in the LLC, and upon L2 cache misses, data can be loaded to L2 caches directly from memory without passing the LLC. Thus, both L2 caches and the LLC become the last line of defense before hitting the memory wall. Cache conflicts must be minimized at both cache layers.

B. Reducing Cache Conflicts

Cache conflicts, or conflict misses, occur in a set associative cache. When data blocks being accessed by programs are not evenly mapped to cache sets, some cache sets are mapped with more data blocks than their capacities. This causes extra misses, which would not be incurred if the data blocks were evenly mapped to cache sets. These extra misses are conflict misses.

The key to reducing cache conflicts is to evenly map data blocks to cache sets. This can be achieved with hardware approaches by improving the mapping of memory addresses to cache sets or with software approaches by improving the mapping of data blocks to memory addresses. Hardware approaches mainly include enhancing cache indexing [29], [30] and adjusting cache associativities [31]. For existing processors and their caches, software approaches try to evenly distribute data blocks onto the memory addresses that can be evenly mapped to cache sets. For system software, this is achieved by allocating non-conflicting pages to the data to be accessed together, as explained in Section I and the next subsection.

C. Memory Management on Virtualized Platforms

On virtualized platforms, the host OS and the guest OS manage memory independently. The host OS allocates host physical pages to hold guest physical pages, which form the physical memory space of each VM. In a VM, the guest OS allocates guest physical pages to hold guest virtual pages, which form the virtual memory space of each workload. Two levels of page tables are used for book-keeping the page allocations and helping with address translations. In the guest OS, normal page tables are used to maintain the mapping between guest virtual pages and guest physical pages. In the host OS, extended page tables are used to maintain the mapping between guest physical pages and host physical pages. With these two levels of page tables, memory allocations may change in each layer without notifying the other layer. For example, upon events such as page faults, swapping, deduplication, and ballooning, the host OS may change the host physical pages allocated to guest physical pages without notifying the guest; the guest OS can reclaim the guest physical pages of a workload and allocate them to another workload without the awareness of the host. Thus, the host physical pages used by a workload are determined by the independent page allocation decisions of the two layers. Neither the guest nor the host has a complete control.

Each OS implements a page placement mechanism with a certain policy (e.g., page coloring or bin-hopping). We use LLC and the page placement mechanism in the host to explain how it functions. Based on the mapping between host physical addresses and the cache sets in LLC, the mechanism divides host physical pages into disjoint groups. The pages in the same group are mapped to the same group of cache sets (e.g., 64 cache sets with a 4KB page size and a 64B cache block size). These cache sets are called a *cache color*. The number of cache colors is determined by the number of cache sets. For example, a LLC with 2048 cache sets has 32 cache colors. With a page placement mechanism, each page is also labelled with a *color*, which is the index of the corresponding cache color. Thus, pages in the same group are in the same color. When allocating pages, a page placement mechanism tries to allocate pages in different colors, such that the data in these pages can be evenly mapped to LLC cache sets. The page placement mechanism in the guest functions in the similar way, except that it uses guest physical addresses and considers the mapping between guest physical pages and virtual LLC cache sets.

App.	Workload description
Img-dnn	Handwriting recognition based on OpenCV [32].
Sphinx	Speech recognition like Apple Siri [33].
Moses	Real time translation like Google translate [34].
Xapian	Search engine used in websites and S/W frameworks [35].
Masstree	In memory K/V store with 50% GET and 50% PUT [36].
Specjbb	Industry-standard JAVA middleware benchmark [37].
Silo	In-memory transactional database with TPCC [38].
RocksDB	Serve requests (random keys,50% SET,50% GET) [39].
PgSql	Join two tables in Postgres [40].
Redis	Serve requests (random keys,50% SET,50% GET) [41].
Memcached	Serve requests (random keys,50% SET,50% GET) [42].
PARSEC	Six benchmarks from PARSEC benchmark suite [43].
SPLASH2X	Four benchmarks from SPLASH2X benchmark suite [44].

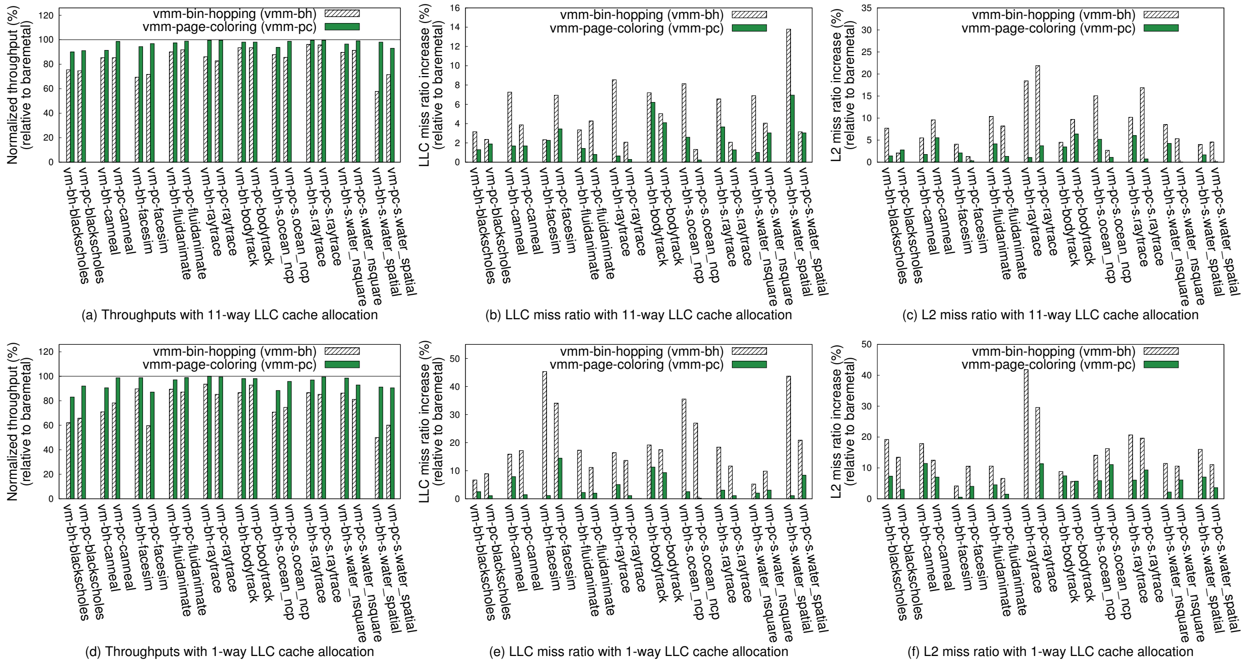
Table I: Programs and workloads used in experiments.

III. Problem Analysis and Motivation

Using experiments, this section demonstrates that workloads suffer increased cache conflicts on virtualized platforms. Our experiments were conducted on a HPE (Hewlett Packard Enterprise) ProLiant DL580 Gen10 server with four Intel Xeon Gold 6138 processors, 256GB memory, two 2TB HDDs, and two 2TB SSDs. Each processor has 20 cores. Each core has a 32KiB L1d cache, a 32KiB L1i cache, and a 16-way 1MiB L2 cache. All the cores in a processor share a 11-way 27.5MiB LLC. We created a virtual machine using KVM/QEMU as the virtual machine monitor (VMM). The virtual machine has one vCPU and 8GiB memory. Both the host OS and the guest OS are Ubuntu Linux 18.04 with kernel updated to 5.3.

We conducted experiments with a diverse set of workloads generated by the benchmarks in Table I. These benchmarks are typical applications from different domains, e.g., AI training, database server, key value store, web and search engine. The first 11 benchmarks (including the first seven benchmarks from TailBench [45]) are latency sensitive. The rest are throughput oriented.

For throughput oriented workloads, we collect their



(a) Throughputs with 11-way LLC cache allocation

(b) LLC miss ratio with 11-way LLC cache allocation

(c) L2 miss ratio with 11-way LLC cache allocation

(d) Throughputs with 1-way LLC cache allocation

(e) LLC miss ratio with 1-way LLC cache allocation

(f) L2 miss ratio with 1-way LLC cache allocation

Figure 1: ***Normalized throughputs and miss ratio increases (LLC and L2 cache) of throughput oriented workloads running in VMs.*** **Throughputs are normalized against those on the host.**

throughputs. For latency sensitive workloads, we collect mean latencies (i.e., average response times) and tail latencies (i.e., longest response times). To help analysis, we also collect the miss ratios of the LLC and L2 cache, because both incur increased memory accesses due to increased cache conflicts.

To highlight the impacts of LLC partitioning, we repeat the experiments for two different LLC space sizes: We first let the workload use the complete LLC (i.e., 11 ways). Then, we let it use only 1 LLC way.

Each of the guest and the host may choose page-coloring or bin-hopping policy in its page placement mechanism[1]. To better understand how the choices of page placement policies affect cache conflicts on virtualized platforms, we test all four different combinations of the two policies at the two layers. Thus, when we run a workload in the guest using a page placement policy (e.g., page coloring), the host may use the same policy (i.e., page coloring) or a different policy (i.e., bin-hopping).

For the same setting (LLC space size and page placement policy), we run each workload on both the host and the guest and compare its performance. For the workload runs on the guest, we run each workload immediately after the VM is booted. Because the workload starts with a clean slate, it can show stable performance across different runs. For ease of presentation and comparison, we normalize the throughput/latency of a workload in the guest agaist the corresponding measurement in the host.

Figure 1 presents the results of the throughput oriented workloads, including their normalized throughputs and how their miss ratios are increased in the LLC and L2 cache when they run in the guest. Figure 2 and Figure 3 present the normalized latencies and miss ratio increases for latency-sensitive workloads for two different LLC space sizes (11 way and 1 way), respectively.

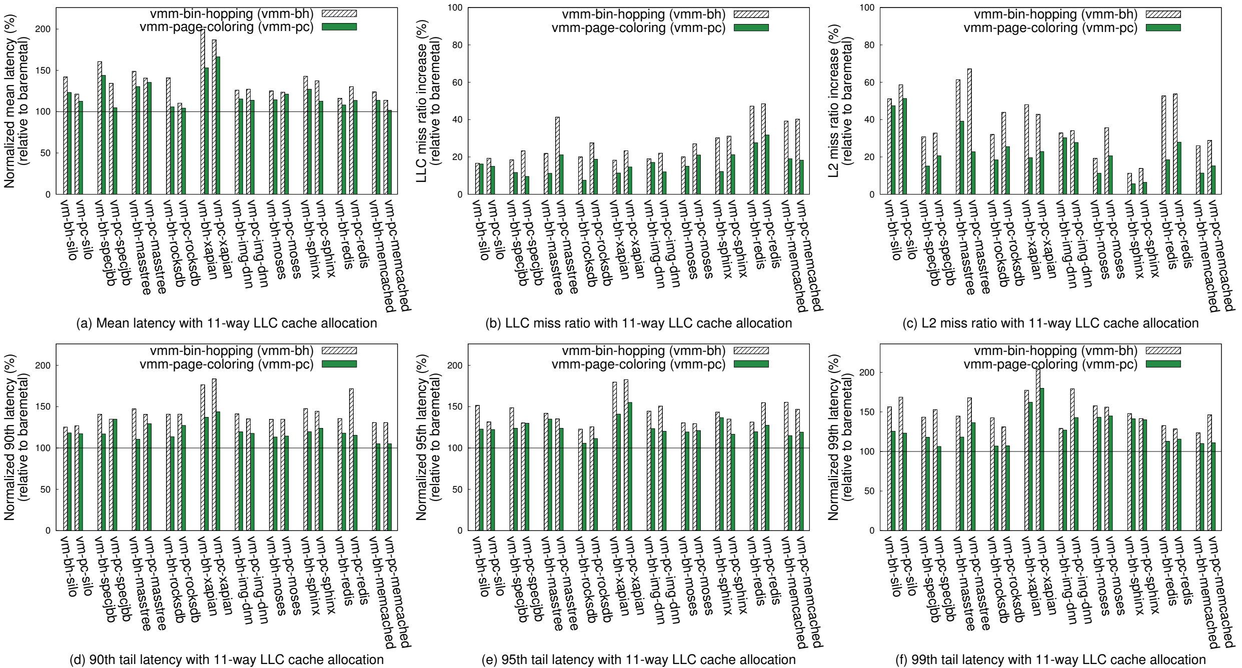
With these figures, the first subsection below shows that page placement mechanisms cannot effectively reduce cache conflicts on virtualized platforms, no matter how different page placement policies are combined. The second subsection shows that cache conflicts become a more serious issue on virtual platforms when workloads use a LLC partition.

A. Increased Cache Conflicts on VMs

As Figure 1(a) shows, all the throughput oriented workloads run slower on virtual machines (9.8% on average), though their performance is sensitive to the choices of page placement policies in the guest and the host. Workload `water_spatial` shows the largest performance degradation (20% on average for 4 combinations of guest/host page placement policies). Interestingly, all the benchmarks perform better when the host uses page-coloring. On average their throughputs are 12.9% higher, compared to their runs when the host uses bin-hopping. The page placement policy choice in the guest does not have an impact as high as the choice in the host.

To understand the performance degradation, Figure 1(b) and Figure 1(c) show the miss ratio increases

[1] For bin hopping, we use the default page placement mechanism implemented in Linux; for page coloring, we implement the FreeBSD's page coloring mechanisms [46] into Linux.



(a) Mean latency with 11-way LLC cache allocation

(b) LLC miss ratio with 11-way LLC cache allocation

(c) L2 miss ratio with 11-way LLC cache allocation

(d) 90th tail latency with 11-way LLC cache allocation

(e) 95th tail latency with 11-way LLC cache allocation

(f) 99th tail latency with 11-way LLC cache allocation

Figure 2: ***Mean latencies, tail latencies, and miss ratio increases (LLC and L2 cache) of latency-sensitive workloads running in VMs without LLC partitioning.*** **Latencies are normalized against those on the host.**

(i.e., miss ratio on the guest - miss ratio on the host) in the LLC and L2 cache, respectively. On average, the LLC miss ratio increase is 3.8%, and the L2 miss ratio increase is 5.6% for these workloads. The LLC miss ratio increase is the highest with workload `water_spatial`. `raytrace` shows the highest L2 miss ratio increase. Thus, it also suffers a large throughput decrease when running in the VM.

These figures also show that LLC miss ratio increases and L2 miss ratio increases are less significant when the host uses page coloring than it uses bin hopping. This indicates that using page coloring in the host helps maintain the effectiveness of the page placement mechanism in the guest. With page coloring, the host chooses the colors of host physical pages in a round-robin manner when allocating host physical pages to contiguous guest physical pages. Thus, when the guest selects and allocates non-conflicting guest physical pages to workloads, it is more likely that the selected guest physical pages are held in non-conflicting host physical pages.

The mean and tail latencies of the latency sensitive workloads in Figure 2 show similar trends. Workload performance is lower in the VM due to increased cache misses. For example, compared to the runs in the host, on average, the mean latency, and 90th, 95th, and 99th tail latencies of the runs in the VM are higher by 29.4%, 31.7%, 34.1%, and 40.2%, respectively. The latencies are also sensitive to the choices of page placement policies. They are lower if the host uses page coloring.

Workloads	**VMM Bin-hopping**	**VMM Page-coloring**	**CoPlace**
`blackscholes`	17.7	8.2	1.7
`water_spatial`	19.9	6.1	1.3
`Xapian`	22.2	4.8	1.1
`Img-dnn`	13.8	7.6	1.6

Table II: ***Conflict levels of four workloads.*** **Conflict levels are the standard deviations of the number of pages mapped to LLC colors.**

To further confirm that the performance degradation is caused by cache conflicts, we select and analyze 2 benchmarks of each type (throughput oriented or latency sensitive), one with a large performance degradation, and the other with an average performance degradation. During the execution of these 4 benchmarks, we periodically sample the memory pages they access and obtain the host physical addresses. Each time, we sample a batch of 7040 pages, the total size of which is equal to the LLC capacity; then we measure whether these pages are evenly mapped to the cache sets in the LLC by counting the number of pages mapped to each LLC color and calculating a standard deviation. We use the average standard deviation across all batches as the *conflict level* of the benchmark.

As shown in Table II, when the host uses bin-hopping, the conflict levels are much higher than when the host uses page coloring. The throughput decrease is the largest with `water_spatial` and the host using bin-hopping (Figure 1). The latency increase is the highest with `Xapian` and the host using bin-hopping (Figure 2). Table II shows that the corresponding conflict levels are also the highest. We have also checked whether the pages are evenly mapped to L2 cache sets and had the similar findings. This indicates that cache conflicts are the cause of increased LLC and L2 miss ratios and performance degradation.

Though the conflict levels are lower with the host using page coloring, there is still much potential to further reduce the levels and thus to improve performance. To illustrate this, we also show the conflict levels with our CoPlace solution in Table II. The conflict levels of all 4 benchmarks are below 2.

These experiments show that existing page placement mechanisms for reducing cache conflicts become less effective, no matter whether they are deployed in the host or the guest. Page placement mechanisms rely on controlling the real physical addresses of application data sets. However, real physical addresses are not available to the page placement mechanism in the VM. At the same time, the page placement mechanism in the host cannot directly control the memory addresses of application data sets. Thus, neither is effective.

B. LLC Partitioning Increases Cache Conflicts

To investigate how LLC partitioning affects cache performance, we created a 1-way partition in the LLC using CAT and assigned it to the workloads. Then we compared the performance degradations and miss ratio increases before and after LLC partitioning for throughput oriented workloads (Figure 1) and latency sensitive workloads (Figure 2 and Figure 3).

As shown in Figure 1, throughput degradation and miss ratio increases (guest executions over host executions) are more significant after LLC partitioning. On average, the throughput degradation is 13.8% with the 1-way LLC partition (Figure 1d)), which is 4% higher than the degradation when the whole LLC is used. LLC miss ratio increases (8.2% on average, Figure 1(e)) are also higher, compared to the increases when the whole LLC is used. With LLC partitioning, the associativity of the LLC space used by the workload is reduced. This decreases the capability of LLC hardware to reduce conflicts and makes it more difficult for the page placement mechanisms in software to reduce LLC conflicts.

It is interesting to observe that L2 cache miss ratio increases (4.8% on average, Figure 1(f)) are also higher after the LLC is partitioned. The L2 cache was not partitioned. More L2 misses are incurred after LLC is partitioned, mainly because increased LLC conflicts raise the space pressure in the L2 cache.

Figure 3 confirms similar performance trends caused by LLC partitioning for latency-sensitive workloads. With LLC partitioning, on average for all workloads, running them in the VM increases mean latencies, 90th-, 95th-, and 99th-tail latencies by 52.9%, 117.2%, 129.3%, and 164.3%, respectively. These increases are much more significant than those without LLC partitioning (Figure 3). With LLC partitioning, the LLC miss ratio increase is 29.2% on average for these workloads, also much higher than that without LLC partitioning.

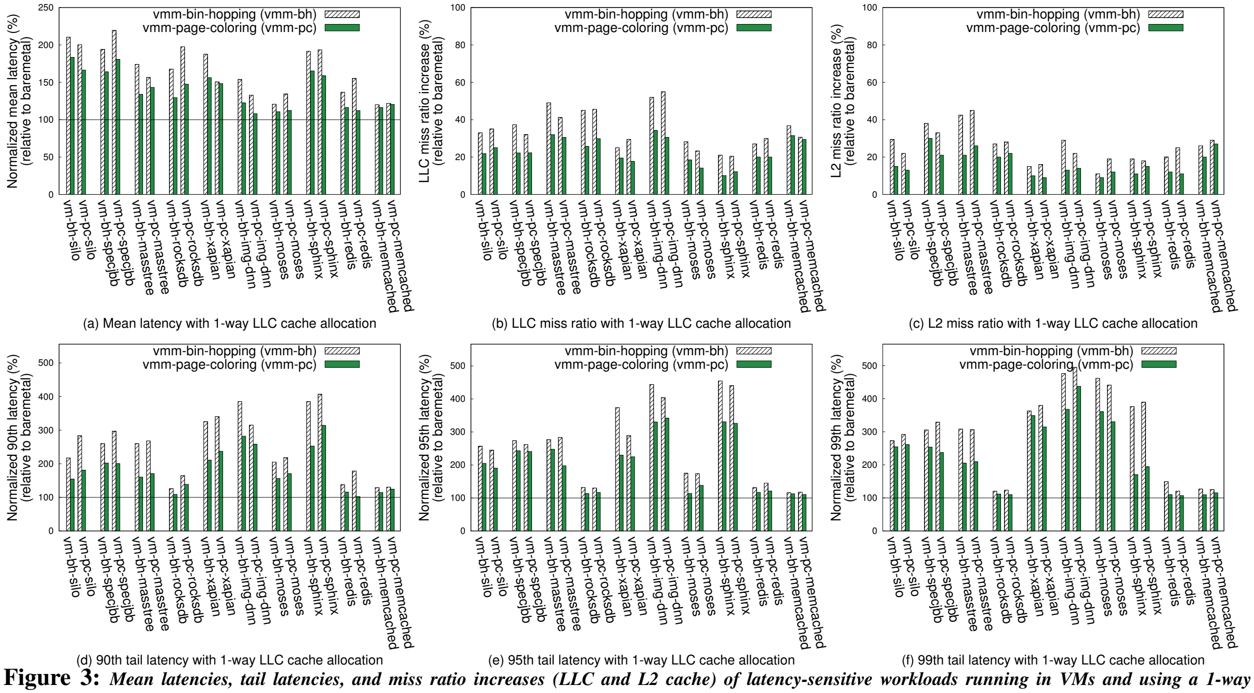
Interestingly, we find that tail latencies are more vulnerable to cache conflicts. This trend is more prominent after LLC partitioning, and can be best illustrated with `Img-dnn`. Running in the VM, the mean latency of `Img-dnn` is increased by 20.6% (without partitioning) and 29.3% (with partitioning). However, its 99th tail latency is respectively increased by 44.6% and 344.1%.

The performance degradation comparison (both throughputs and latencies) also indicates that page placement mechanisms become more important after LLC partitioning. With the 1-way LLC partition, the performance advantage of using page coloring over using bin-hopping in the host becomes more prominent than that without LLC partitioning. For example, without LLC partitioning, replacing bin-hopping with page coloring in the host can improve throughputs by 12.9% for throughput oriented workloads; after cache partitioning, this number increases to 19.8%. For latency sensitive workloads, without LLC partitioning, replacing bin-hopping with page coloring can reduce mean latencies and 95th tail latencies by 21.6% and 18.5%, respectively. However, with LLC partitioning, these numbers increase to 28.7% and 56.4%. By comparing the miss ratio increases before and after LLC partitioning, we find that replacing bin-hopping with page coloring in the host can more effectively reduce LLC and L2 cache misses after LLC partitioning.

While the above observation suggests to improve the page placement mechanism in the host to reduce cache conflicts, further improving the design of the page coloring mechanism does not seem to be an effective solution. Using page coloring in the host achieves better performance than using bin-hopping, because this makes the page placement mechanism in the guest slightly more effective in identifying conflicting and non-conflicting pages. However, page coloring is not specifically designed for this purpose. Thus, for most pages, the page placement mechanism in the guest still cannot correctly determine whether they are conflicting or non-conflicting pages.

This can be indicated by comparing workload performance before and after switching the page placement policy in the guest (page coloring to bin-hopping or vice versa). Switching page placement policies in the guest can hardly impact performance, no matter whether the LLC is partitioned or not. However, when running in the host, where conflicting and non-conflicting pages can be correctly identified, most workloads achieve better performance with bin-hopping than with page coloring. For example, the average throughput is 6.1% higher with bin-hopping for throughput oriented workloads.

This motivates us to design CoPlace as a novel



(a) Mean latency with 1-way LLC cache allocation

(b) LLC miss ratio with 1-way LLC cache allocation

(c) L2 miss ratio with 1-way LLC cache allocation

(d) 90th tail latency with 1-way LLC cache allocation

(e) 95th tail latency with 1-way LLC cache allocation

(f) 99th tail latency with 1-way LLC cache allocation

Figure 3: ***Mean latencies, tail latencies, and miss ratio increases (LLC and L2 cache) of latency-sensitive workloads running in VMs and using a 1-way LLC partition.*** **Latencies are normalized against those on the host.**

approach to effectively reduce cache conflicts in modern virtualized clouds, which is presented below.

IV. CoPlace Design and Implementation

As shown in §III, on virtualized platforms, the page placement mechanisms in the guest and host OSs cannot effectively reduce cache conflicts, due to two layers of uncoordinated memory management and the semantic gap produced by virtualization. For virtualized clouds, it is not possible to reduce one layer of memory management. Thus, the issue of increased cache conflicts must be addressed by boosting the synergy between the page placement mechanisms in the guest and the host.

In existing systems, the page placement mechanisms in the guest and the host function independently. Thus, an intuitive design to boost the synergy is to enhance both of them to make them collaborate. However, this design requires significant interaction between the guest and the host OSs, in order to make coherent or complementary page placement decisions. The interaction may involve costly context switches between the guest and the host. At the same time, it may need to change the guest OS and/or the interface between the guest and the host. Such changes are undesirable, since they reduce the portability of the solution.

We recognize that a practical and portable solution must follow three principles: 1) the guest OS and the host OS should still be able to allocate and reclaim memory pages without notifying each other; 2) the interface between the guest and the host must not be changed; 3) the guest OS must not be changed. The first principle is to maintain two layers of memory management required by virtualization and to minimize the overhead at the same time. The other two principles are to maximize the portability of the solution.

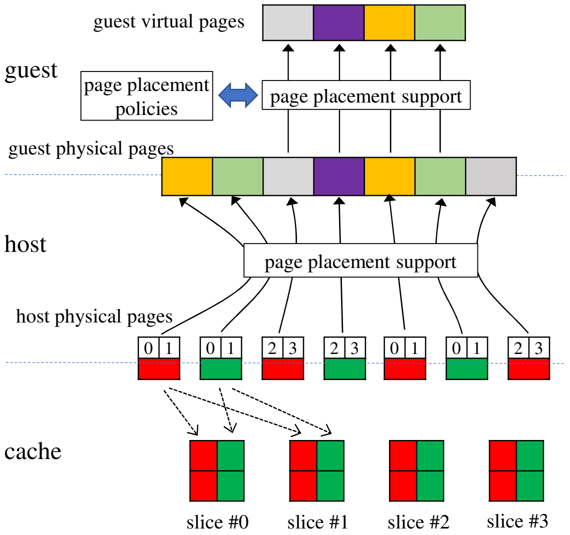


Figure 4: CoPlace Overview. Numbers in host physical pages are cache slice indexes that the pages are mapped to.

Following these principles, we design CoPlace. Its overall architecture is as shown in Figure 4. Since the guest OS (shown with the top section in the figure) is not changed, its page placement mechnism is fully-fledged with two components. One component implements *page placement policies* to make page placement decisions (e.g., page coloring or bin-hopping). It selects page colors upon page allocation requests, such that application data sets can be evenly mapped to the cache colors in the virtual LLC. The other component provides *page* *placement support* to enforce page placement decisions, i.e., allocating pages in designated colors, as shown with 4 colors of pages in the figure. It determines page colors using guest physical addresses.

The guest page placement mechanism cannot reduce cache conflicts in real caches. However, its decisions about how to distribute application data sets are directly guided by workloads and are adequate. They must be well leveraged in the solution. Its regular assignments of guest physical pages based on the decisions can also be leveraged, such that the solution does not need to tackle the allocation of guest physical pages and can focus only on how to allocate host physical pages.

CoPlace fully leverages the guest page placement mechanism and make it effective by getting the synergy from the host. It ensures that, for the guest physical pages deemed by the guest to be in different colors, they are really mapped to different cache sets in the physical LLC. This is achieved by making the *page placement support component* in the host select host physical pages carefully based on the structure of physical caches.

Figure 4 illustrates this using a LLC with 4 slices. Each slice has two ways, 4KB each. The data blocks in each page are evenly divided and mapped into two slices, as shown with the dotted arrows. In each slice, the cache sets that can hold the data blocks from the same page are also called a *cache color* (e.g., red or green). Thus, pages mapped to the same cache color in the same slices are conflicting pages (e.g., the first and fifth pages mapped to red cache colors in slices 0 and 1); and slice indexes and cache color together form a color that labels a host physical page. The host physical pages in the figure are in 4 different colors. For the guest physical pages in different colors, their host physical pages are also in different colors.

Some processors (e.g., Intel Scalable CPUs [47]) use undisclosed hash functions to map the data blocks in a page to a few possible cache slices [48], [49]. Thus, slice indexes cannot be precisely determined. To label a host physical page with a color, we use cache color and some of the bits in the page address that are used to determine cache slices. (Inside a slice, the data blocks of the page are mapped to a cache color in a traditional way [50]–[52].) Among the bits determining cache slices, we select the least significant bits, because pages with different values in these bits (e.g., base pages in a huge page) are usually mapped by hardware to different cache slices to minimize cache conflicts.

For the processors with non-inclusive LLCs, it is crucial to mitigate cache conflicts for both L2 cache and LLC. Thus, we use the combination of the cache color in L2 and the cache color in LLC as the colors of host physical pages.

When allocating a host physical page to a guest physical page, CoPlace must select a host physical page in a particular color. This requirement can reduce the effectiveness of memory deduplication (e.g., kernel same page merging in Linux/KVM) [53], [54] and memory ballooning [53]. For memory deduplication, with this requirement, the host physical pages holding the guest physical pages in different colors cannot be merged even when they have identical contents. To address this issue, CoPlace allows the memory deduplication component to disregard this requirement when identical pages are scarce. For memory ballooning, due to this requirement, the host physical pages released from one VM may not be used by another VM if their colors are undesirable. To address this issue, CoPlace reclaims pages evenly in all the colors. These enhancements are named `eKSM` and `eBal` in CoPlace (not shown in Figure 4).

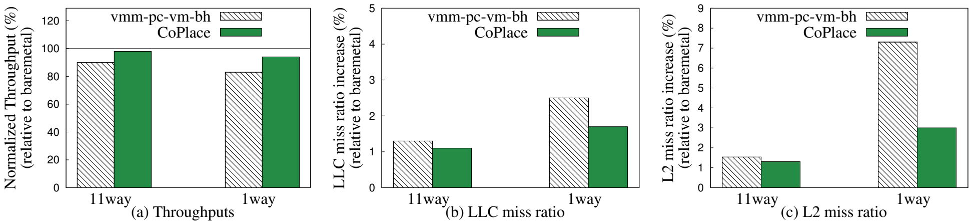
V. Evaluation

We have implemented CoPlace based on Linux KVM. We added/modified 694 lines of source code mainly in the Linux memory manager component, KSM [54], and the virtio ballooning driver [55], [56]. With the prototype implementation, we test CoPlace by running benchmarks in a virtual machine.

A. Experimental Settings

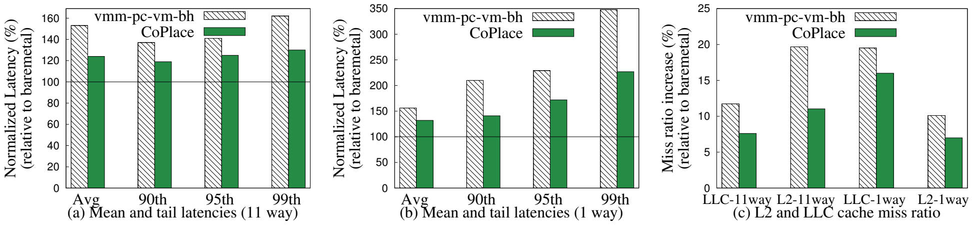
Based on the measurements in §III, benchmarks show higher performance on a system with the host using page coloring than that on the vanilla system, where the host uses bin-hopping. Thus, we choose the system with the host using page coloring as the baseline system. We compare the performance of the benchmarks on the system with CoPlace prototype against their performance on the baseline system. For both CoPlace and the baseline system, bin-hopping is used in the guest OS. Refer to §III for detailed system configurations.

We test CoPlace using constant and varying workloads. A constant workload is generated by running a benchmark immediately after the VM is launched, and thus shows stable performance across different runs. More importantly, on the baseline system, the benchmark usually can achieve better performance when the system is just booted up than later after some other applications finish their executions on the VM. When a VM is just launched, some of its guest physical pages have not be allocated with host physical pages. When the benchmark saves data into these pages, page faults are caused at the host level, and the host OS needs to allocate host physical pages to save the data. On the baseline system, with the page coloring page placement mechanism, the host can allocate non-conflicting pages. This helps reduce cache conflicts caused by accessing the data. This benefit persists as long as the page still holds the data, and thus helps improve the performance of workloads, particularly throughput oriented workloads, such as scientific computing applications using



(a) Throughputs (b) LLC miss ratio (c) L2 miss ratio

Figure 5: ***Normalized throughput and miss ratio increases (LLC and L2 cache) of*** `blackscholes` ***on the baseline system (vmm-pc-vm-bh) and*** **CoPlace.**



(a) Mean and tail latencies (11 way) (b) Mean and tail latencies (1 way) (c) L2 and LLC cache miss ratio

Figure 6: ***Mean and tail latencies of*** `Xapian` ***on the baseline system (vmm-pc-vm-bh) and*** **CoPlace.**

static arrays. The benefit reduces when some of the pages are reclaimed or are used to hold other data. This usually occurs in service oriented applications, which use dynamic data structures to save temporary data.

A varying workload is generated by running a benchmark after the completion of another benchmark with a large working set. During the execution of the earlier benchmark, a large amount of memory is allocated to the VM to hold the data set of the benchmark. When the earlier benchmark finishes, the memory becomes free memory in the guest OS. But the host does not reclaim the memory, because it is not under memory pressure. Thus, during the execution of the benchmark under test, the guest OS can allocate memory to the benchmark without incurring any page faults at the host level. (The benchmark under test has a smaller working set size than the earlier benchmark.) Therefore, the performance benefit observed with a constant workload is not available to a varying workload. Even worse, the performance of the benchmark may be negatively impacted by the execution of the earlier benchmark, as we will show in the evaluation, and may change if another benchmark was selected to run first.

We use constant and varying workloads to create three different scenarios. A constant workload generated by a scientific computing application (`blackscholes`) represents a scenario in favor of the baseline system. A constant workload generated by a service oriented application (`Xapian`) represents a scenario that is neutral to the baseline system. A varying workload that is negatively impacted by an earlier workload represents an unfavorable scenario for the baseline system. We show that CoPlace outperforms the baseline system in all these scenarios. We also test how well CoPlace, specifically its enhanced KSM (`eKSM`) and enhanced memory ballooning(`eBal`), deals with multiple VMs.

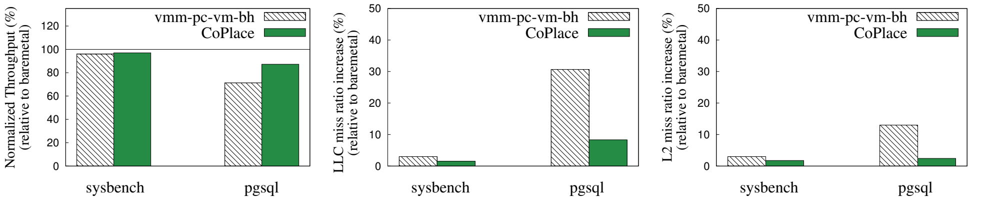
B. Experiments with Constant Workloads

We first select `blackscholes` from PARSEC benchmark suite to generate a constant workload. `blackscholes` computes the price of an array of options analytically using Black-Scholes Partial Differential Equation (PDE). Figure 5(a) compares its throughput on CoPlace and the baseline system. The throughput is normalized to that on the host (vanilla Linux). With CoPlace, the throughput of `blackscholes` is 8.8% and 13.3% higher than it on the baseline system for the two scenarios using the whole LLC and 1 LLC way, respectively. With CoPlace, the performance of `blackscholes` is only 4% lower than that on the host on average for the two scenarios. This is partly caused by the overhead introduced by the second level memory address translation. As shown in Figure 5(b) and Figure 5(c), though CoPlace can effectively reduce LLC and L2 cache misses, the miss ratios are still higher than those on the host. This may be caused by the guest OS and other system services running in the guest.

We use `Xapian` from TailBench to generate another constant workload. `Xapian` is an open source search engine that is widely used in websites (e.g., the Debian wiki) and software frameworks (e.g., Catalyst). Compared to `blackscholes`, the performance of `Xapian` is improved by larger percentages with CoPlace. As shown in Figure 6, when LLC is not partitioned, CoPlace can reduce the mean, 90th, 95th, and 99th latencies by 29.3%, 18.1%, 16.9%, and 32.3%, respectively; when the 1-way LLC partition is used, CoPlace can reduce these latencies by 24.0%, 69.4%, 57.2%, and 121%, respectively. With CoPlace, the performance of `Xapian` is still substantially lower than it on the host, because `Xapian` is an operating system intensive workload sufferring significant virtualization overhead, including extra cache misses introduced by the operations in the extra layer of operating system.

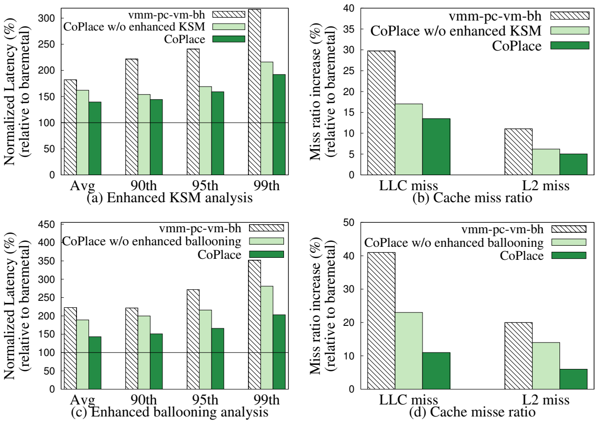
C. Experiments with Varying Workload

We use SysBench and `PgSql` to generate a varying workload. SysBench reads data from a large file into an array. `PgSql` executes SQL commands in a PostgreSQL



(a) Throughputs with 11-way LLC cache allocation (b) LLC miss ratio with 11-way LLC cache allication (c) L2 miss ratio with 11-way LLC cache allocation

Figure 7: ***Normalized throughputs, and miss ratio increases in LLC and L2 cache when*** `PgSql` ***runs after SysBench in a VM.*** **LLC is not partitioned.**



(a) Enhanced KSM analysis (b) Cache miss ratio

(c) Enhanced ballooning analysis (d) Cache misse ratio

Figure 8: ***Performance of*** `Xapian` ***when it runs on the baseline system,*** **CoPlace** ***without*** `eKSM` ***or*** `eBal`, ***and*** **CoPlace.** `Xapian` **uses a 1-way LLC partition.**

DBMS. The SQL commands join a small table (5MB) and a large table (500MB) using table-join operations. Since SysBench is I/O intensive, we are more interested in the performance of `PgSql`.

Before we run the varying workload, we have tested how the performance of `PgSql` is affected by SysBench. When `PgSql` runs after SysBench, its performance is 23.6% lower than it running with a clean slate on a just booted VM. The reason is as follows. When SysBench reading file data into an array, two types of guest virtual pages, the pages for buffering the file data in guest OS (OS buffer cache) and the pages for saving the array, are requested from the guest OS in an interleaved manner. However, the bin-hopping mechanism and the buddy memory allocator in the guest allocate contiguous guest physical pages, and then the host allocates contiguous host physical pages to hold these guest physical pages upon page faults. Because the two types of guest virtual pages are requested in an interleaved manner, each type of the pages may dominate some LLC colors. When SysBench terminates, the guest physical pages for the array are reclaimed in the VM; but the memory pages in OS buffer cache are not. Thus, when `PgSql` runs, the previously reclaimed guest physical pages are reallocated to `PgSql`, causing its pages unevenly distributed in different LLC colors. We have checked the conflict level of `PgSql`, which is as high as 50.2. This causes increased cache misses and degrades the performance of `PgSql`.

Then, in the VM, we run SysBench followed by `PgSql` on both CoPlace and the baseline system. Figure 7 shows the normalized throughputs and miss ratio increases of SysBench and `PgSql` when they run in the VM, relative to the executions on the host. LLC is not partitioned. Compared to the baseline system, CoPlace can substantially reduce the misses in the LLC and the L2 cache. Thus, with CoPlace, the throughput of `PgSql` is 22.5% higher than it on the baseline system.

D. Dealing with Multiple VMs

The experiments above only test the capability of CoPlace on reducing cache conflicts on a single VM. This subsection tests how well CoPlace, specifically its enhanced KSM (`eKSM`) and enhanced memory ballooning (`eBal`), deals with multiple VMs. `eKSM` and `eBal` improve existing memory deduplication and memory ballooning mechanisms, such that the page placement decisions made in the guest can be well maintained.

To test `eKSM`, we launched 11 VMs on the same host. Each VM runs an instance of `Xapian`. Each VM has one vCPU on a dedicated CPU core and uses a 1 way LLC partition. We compare the performance of three systems under workload `Xapian`: the baseline system (`VMM-PC-VM-BH`), CoPlace with `eKSM` disabled, and CoPlace. Figure 8(a) shows the mean and tail latencies, relative to 11 `Xapian` instances running on the host. Though CoPlace without `eKSM` can effectively reduce the latencies of `Xapian`, by making memory deduplication "cache conflicts aware", `eKSM` can futher reduce reduce mean latency (22.6%) and tail latencies (9.6% for 90th, 9.9% for 95th, and 23.9% for 99th tail latency).

To test the effectiveness of `eBal`, we examine how `Xapian` performs when about half of the memory pages used by it are obtained indirectly through memory ballooning. In a VM, we run `Xapian` together with a memory hog application, which consumes almost all the free memory space in the VM by creating arrays and saving data. To avoid the interference from the memory hog, we use a two-vCPU VM, pin the vCPUs on two cores, and run `Xapian` and the memory hog on different vCPUs. At the same time, we allocate a 1-way LLC partition for each of them. To make `Xapian` use some memory pages obtained through ballooning, we assign the VM another 20MB memory space (roughly equal to the working set size of `Xapian`) using memory ballooning. Since Xapian requests and frees memory dynamically, some of the memory pages will be gradually allocated to Xapian by the guest OS.

Figure 8(d) confirms that eBal increases COPLACE's capability to reduce LLC and L2 misses. This is because eBal can ensure that the memory pages in a VM can be evenly mapped to different cache colors. Thus, COPLACE can be more effective in reducing cache conflicts. For Xapian, eBal helps COPLACE further reduce mean latency (45.9%) and tail latencies (48.9% for 90th, 50% for 95th, and 78% for 99th), as shown in Figure 8(c).

VI. Related Work

Reducing cache conflicts. Extensive research has been focused on reducing cache conflicting. Both hardware and software solutions have been developed, as elaborated in Section II. Minimizing cache conflicts requires the combined efforts from all system layers. COPLACE is a solution at the system software layer, and is complementary to the solutions at other system layers. Compared to other solutions at the system software layer, such as page coloring and bin-hopping, COPLACE is designed specifically to virtualized platforms.

Reducing cache interference with partitioning. As an effective method to reduce cache interference, cache partitioning has attracted much attention. Page coloring based software approaches divide cache sets between workloads [8], [9], [14], [57], [58]. They can reduce cache interference without increasing cache conflicts, because cache associativity is not reduced. However, they incur high overhead when adjusting partition sizes [57]. Hardware approaches usually partition cache ways [59], including Intel cache allocation technology (CAT) and AMD cache allocation enforcement (CAE) [3]–[5]. The latest efforts focus on improving software systems to better utilize CAT [10], [11], [16], [17], [60]. As the paper shows, hardware approaches increase cache conflicts. This paper presents COPLACE to effectively reduce them in virtualized systems.

vCache provides transparent and isolated virtual LLCs (vLLC) for the workloads in VMs [6]. It designs architectural support to allow the guest OS to index the LLC using guest physical addresses. Thus, various cache optimization techniques can be performed in the guest OS, including LLC partitioning using page coloring and page placement mechanisms. Nevertheless, there are three main issues with vCache that may limit its effectiveness and adoption. First, virtual-indexed caches may have inferior performance than real-indexed cache due to virtual address space changes (e.g., context switches) [1], [61]–[64]. Second, vCache requires hardware changes, which would not be available in near future. The last but not least, vCache only targets LLC; with vCache, L1 and L2 caches are still indexed with host physical addresses. Thus, cache optimization techniques for L2 caches may still be ineffective. This limits its application on modern systems with non-inclusive cache hierarchy.

Huge pages. Huge pages are mainly used to reduce TLB misses [25], [65]–[68]. They can also be used to reduce cache conflicts on virtualized platforms. In a huge page allocated to a VM, host physical addresses and guest physical addresses share the same offsets within the page. Because huge pages are large enough, the offsets within the pages contain the memory address bits used to determine cache sets. With the offsets, the page placement mechanism in the guest OS can effectively reduce cache conflicts.

However, there are three issues with using huge pages to reduce cache conflicts: 1) huge pages can significantly degrade the performance of user applications; and thus they are usually disabled in many software systems [68]–[80]; 2) huge pages are not well supported by cloud providers such as AWS EC2 [81]; 3) memory fragmentation is a well known problem associated with using huge pages [25], [82]–[85]. Previous work has shown that memory can be quickly fragmented in multi-tenant clouds [86]. When memory is fragmented, systems usually synchronously compact memory with a page fault handler. This increases average and tail latencies of latency critical workloads.

VII. Conclusion and Future Work

With the prevalence of modern processors in clouds, emerging hardware extensions for cache allocation (e.g., Intel CAT) are being widely used to mitigate performance problems caused by cache interference between workloads. However, with these extensions, cache conflicts in L2 and last level caches become a serious issue in virtualized clouds, and this issue is barely noticed. By identifying and analyzing this issue, the paper tries to bring the attention of the community to this issue. The paper also presents COPLACE as an effective system solution with low overhead and high portability. Extensive experiments confirm its effectiveness.

As future work, we want to adapt and test COPLACE for the systems with non-volatile memory (NVM), such as Intel Optane DC Persistent Memory Module. When NVM is used in the memory mode, DRAM acts as its direct-mapped L4 cache. In virtualized clouds, similar cache conflict problems are expected to occur in the DRAM L4 cache. We believe COPLACE can be an effective solution.

Acknowledgments

We thank the reviewers for their feedback. This work is partially funded by US National Science Foundation grant CCF 1617749.

References

[1] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-index caches," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 338–359, Nov. 1992.

[2] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 14:1–14:13. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190555

[3] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "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)*. IEEE, 2016, pp. 657–668.

[4] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "Cachescouts: Fine-grain monitoring of shared caches in cmp platforms," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 339–352.

[5] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004, pp. 257–266.

[6] D. Kim, H. Kim, N. S. Kim, and J. Huh, "vCache: Architectural support for transparent and isolated virtual llcs in virtualized environments," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, 2015, pp. 623–634.

[7] D. Kim, H. Kim, and J. Huh, "vcache: Providing a transparent view of the llc in virtualized environments," *IEEE Computer Architecture Letters*, vol. 13, no. 2, pp. 109–112, 2014.

[8] H. Kim and R. Rajkumar, "Real-time cache management for multi-core virtualization," in *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, 2016, pp. 1–10.

[9] H. Kim and R. R. Rajkumar, "Predictable shared cache management for multi-core real-time virtualization," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 1, December 2017.

[10] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 450–462.

[11] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 13.

[12] L. Funaro, O. A. Ben-Yehuda, and A. Schuster, "Ginseng: Market-driven {LLC} allocation," in *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016, pp. 295–308.

[13] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem, "Ginseng: Market-driven memory allocation," in *ACM SIGPLAN Notices*, vol. 49, no. 7. ACM, 2014, pp. 41–52.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 367–378.

[15] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2018, p. 5.

[16] J. Park, S. Park, and W. Baek, "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*, 2019, pp. 1–16.

[17] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*. IEEE, 2017, pp. 211–222.

[18] M. Hocko and T. Kalibera, "Evaluating the impact of page coloring and bin hopping on performance nondeterminism of linux applications."

[19] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 244–255, 1996.

[20] T. Sherwood, B. Calder, and J. Emer, "Reducing cache misses using hardware and software page placement," in *Proceedings of the 13th international conference on Supercomputing*, 1999, pp. 155–164.

[21] "target-i386: present virtual L3 cache info for vcpus," 2016. [Online]. Available: https://lists.gnu.org/archive/html/qemu-devel/2016-08/msg03956.html

[22] "Amazon EC2 High Memory Instances," https://aws.amazon.com/ec2/instance-types/high-memory/).

[23] A. Panwar, R. Achermann, A. Basu, A. Bhattacharjee, K. Gopinath, and J. Gandhi, "Fast local page-tables for virtualized numa servers with vmitosis," in *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[24] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, “Mitosis: Transparently self-replicating page-tables for large-memory machines,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 283–300.

[25] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, “Coordinated and efficient huge page management with ingens,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 705–721.

[26] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[27] D. Ojika, B. Patel, G. A. Reina, T. Boyer, C. Martin, and P. Shah, “Addressing the memory bottleneck in ai model training,” 2020.

[28] B. Chen, T. Medini, J. Farwell, C. Tai, A. Shrivastava *et al.*, “Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 291–306, 2020.

[29] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, “Using prime numbers for cache indexing to eliminate conflict misses,” in *10th International Symposium on High Performance Computer Architecture (HPCA’04)*, 2004, pp. 288–299.

[30] A. González, M. Valero, N. Topham, and J. M. Parcerisa, “Eliminating cache conflict misses through xor-based placement functions,” in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 76–83.

[31] M. K. Qureshi, D. Thompson, and Y. N. Patt, “The v-way cache: demand-based associativity via global replacement,” in *32nd International Symposium on Computer Architecture (ISCA’05)*. IEEE, 2005, pp. 544–555.

[32] “A deep network handwriting classifier,” https://github.com/xingdi-eric-yuan/multi-layer-convnet.

[33] W. Walker, P. Lamere, P. Kwok, B. Raj, R. Singh, E. Gouvea, P. Wolf, and J. Woelfel, “Sphinx-4: A flexible open source framework for speech recognition,” 2004.

[34] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens *et al.*, “Moses: Open source toolkit for statistical machine translation,” in *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, 2007, pp. 177–180.

[35] “Xapian project,” https://github.com/xapian/xapian.

[36] Y. Mao, E. Kohler, and R. T. Morris, “Cache craftiness for fast multicore key-value storage,” in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 183–196.

[37] “A deep network handwriting classifier,” https://www.spec.org/jbb2015/.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, “Speedy transactions in multicore in-memory databases,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 18–32.

[39] “RocksDB NoSQL Storage System,” https://rocksdb.org/. [Online]. Available: https://rocksdb.org/

[40] “Comparisons of joins between MongoDB and PostgreSQL,” https://www.enterprisedb.com/blog/comparison-joins-mongodb-vs-postgresql.

[41] “Redis In-memory Key-Value Database,” http://redis.io/.

[42] https://memcached.org/.

[43] “The Princeton application repository for shared-memory computers (PARSEC),” http://parsec.cs.princeton.edu/, 2010.

[44] “SPLASH2X Benchmark Suite,” http://parsec.cs.princeton.edu/parsec3-doc.htm, 2017.

[45] H. Kasture and D. Sanchez, “Tailbench: a benchmark suite and evaluation methodology for latency-critical applications,” in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[46] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[47] “Intel Xeon Scalable Processors,” https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html.

[48] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić, “Make the most out of last level cache in intel processors,” in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[49] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+ abort: A timer-free high-precision l3 cache attack using intel {TSX},” in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 51–67.

[50] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, “Mapping the intel last-level cache.” *IACR Cryptology ePrint Archive*, vol. 2015, p. 905, 2015.

[51] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, “Attack directories, not caches: Side channel attacks in a non-inclusive world,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.

[52] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Systematic reverse engineering of cache slice selection in intel processors,” in *2015 Euromicro Conference on Digital System Design*. IEEE, 2015, pp. 629–636.

[53] C. A. Waldspurger, “Memory resource management in vmware esx server,” in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’02. USA: USENIX Association, 2002, p. 181–194.

[54] “Kernel Samepage Merging,” https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[55] R. Russell, “virtio: towards a de-facto standard for virtual i/o devices,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[56] “QEMU/KVM automatic ballooning support,” https://www.linux-kvm.org/page/Projects/auto-ballooning.

[57] X. Zhang, S. Dwarkadas, and K. Shen, “Towards practical page coloring-based multicore cache management,” in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 89–102.

[58] X. Ding, K. Wang, and X. Zhang, “Ulcc: a user-level facility for optimizing shared cache performance on multicores,” in *Acm sigplan notices*, vol. 46, no. 8. ACM, 2011, pp. 103–112.

[59] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 423–432.

[60] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, “dcat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service,” in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 14.

[61] R. L. Sites and A. Agarwal, “Multiprocessor cache analysis using atum,” *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2, pp. 186–195, 1988.

[62] G. Taylor, P. Davies, and M. Farmwald, “The tlb slice — a low-cost high-speed address translation mechanism,” in *Proceedings of the 17th annual international symposium on Computer Architecture*, 1990, pp. 355–363.

[63] W.-H. Wang, J.-L. Baer, and H. M. Levy, “Organization and performance of a two-level virtual-real cache hierarchy,” *ACM SIGARCH Computer Architecture News*, vol. 17, no. 3, pp. 140–148, 1989.

[64] A. J. Smith, “Cache memories,” *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[65] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee, “Proactively breaking large pages to improve memory overcommitment performance in vmware esxi,” in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2015, pp. 39–51.

[66] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, “Smartmd: A high performance deduplication engine with mixed pages,” in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 733–744.

[67] J. Navarro, S. Iyer, P. Druschel, and A. Cox, “Practical, transparent operating system support for superpages,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 89–104, 2002.

[68] A. Panwar, S. Bansal, and K. Gopinath, “Hawkeye: Efficient fine-grained os support for huge pages,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 347–360.

[69] “Cloudera reports huge pages seriously degrade performance of Hadoop workloads,” https://docs.cloudera.com/cdp-private-cloud-base/7.1.3/managing-clusters/topics/cm-disabling-transparent-hugepages.html.

[70] “Recommendation for disabling huge pages for Redis,” https://redis.io/topics/latency.

[71] “MongoDB does not recommend enabling transparent huge pages,” https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/.

[72] “Huge pages are disabled by SAP HANA on Azure,” https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/sap/hana-monitor-troubleshoot.

[73] “Recommendation for disabling huge pages for VoltDB,” https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[74] “Huge pages must be disabled in Couchbase,” https://docs.couchbase.com/server/5.5/install/thp-disable.html.

[75] “IBM recommends turning off huge pages due to high CPU utilization,” http://www-01.ibm.com/support/docview.wss?uid=swg21677458.

[76] “High CPU utilization in Mysql due to transparent huge pages,” http://developer.okta.com/blog/2015/05/22/tcmalloc.

[77] “High CPU utilization in Hadoop due to transparent huge pages,” https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad/.

[78] “Recommendation for disabling huge pages for NuoDB,” https://nuodb.com/blog/linux-transparent-huge-pages-jemalloc-and-nuodb.

[79] “Using KSM can reduce the occurrence of transparent huge pages,” https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-memory-tuning.

[80] “Large-page support in windows,” https://docs.microsoft.com/en-us/windows/win32/memory/large-page-support?redirectedfrom=MSDN.

[81] “Huge pages are not well supported by AWS EC2,” https://forums.aws.amazon.com/thread.jspa?messageID=477265.

[82] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, “Tradeoffs in supporting two page sizes,” in *Proceedings of the 19th annual international symposium on Computer architecture*, 1992, pp. 415–424.

[83] N. Ganapathy and C. Schimmel, “General purpose operating system support for multiple page sizes.” in *USENIX Annual Technical Conference*, no. 98, 1998, pp. 91–104.

[84] I. Subramanian, C. Mather, K. Peterson, and B. Raghunath, “Implementation of multiple pagesize support in hp-ux.” in *USENIX Annual Technical Conference*, 1998, pp. 105–119.

[85] A. Margaritov, D. Ustiugov, A. Shahab, and B. Grot, “Ptemagnet: Fine-grained physical memory reservation for faster page walks in public clouds,” in *The 26th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, 2021.

[86] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker, “Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure,” in *Proceedings of the Middleware 2011 Industry Track Workshop*, 2011, pp. 1–7.