Reestablishing Page Placement Mechanisms for Nested Virtualization

Xiaowei Shang, Weiwei Jia, Jianchen Shan, Xiaoning Ding, and Cristian Borcea

Abstract—Page placement mechanisms have long been used to reduce cache conflict misses. They become more important in clouds where the emerging way-based cache partitioning is used for better workload isolation but at a cost of increased cache conflicts. However, page placement mechanisms become ineffective in virtualized environments, such as clouds, because the real locations of memory pages (i.e., their host physical addresses) are hidden from guest OSs. The paper proposes xPLACE as a solution to reestablish page placement mechanisms under the nested virtualization configuration. To keep high portability and low overhead, xPLACE follows an approach that creates a synergy between the host and guest VMs, such that the page placement mechanism inside each guest VM becomes effective even if its page placement decisions are made based on the guest physical addresses of memory pages. The paper addresses the technical issues for implementing this approach in the nested virtualization setting, particularly how to create the synergy with the obstacle created by guest hypervisors sitting between the host and guest VMs. Evaluation based on the prototype implementation and diverse real world applications shows that xPLACE can greatly reduce cache conflicts and improve application performance in the nested environment.

Index Terms—Cache Conflicts, Nested Virtualization, Multi-Core, Memory Management, Page Coloring, Page Placement

1 Introduction

Recently, it has become mainstream to manage last-level cache (LLC) space using new CPU hardware supports, such as Intel cache allocation technology (CAT) [1], [2], [3]. This is particularly important for cloud platforms, where the performance isolation between workloads is critical, but may be destroyed through the sharing of LLC space. With these hardware supports, system software can perform way-based LLC partitioning (i.e., different cache partitions containing different cache ways) and assign different workloads with different LLC partitions [1], [4].

Though way-based LLC partitioning mitigates the interference between workloads, it increases cache conflicts (a.k.a. conflict cache misses). It turns a high-associativity LLC into multiple low-associativity partitions. Cache conflicts increase because the associativity reduces. (Fully associative caches yield no conflict misses.)

Native systems rely on page placement mechanisms (e.g., page coloring and bin hopping) to reduce cache conflicts [5]. Page placement mechanisms (PPMs) try to map virtual pages to different cache sets in LLC, such that the accesses to the data in these pages will hit different LLC sets, and will not cause conflicts. This is achieved by controlling the mapping (i.e., placement) of virtual pages to physical pages and leveraging the fixed mapping between physical memory pages and the sets in LLC. PPMs are implemented in operating systems where the placement of virtual pages is achieved via the careful allocation of physical pages to virtual pages. Specifically, they extract "page colors" from physical page addresses based on the set indexing of LLC. Physical pages with different colors are mapped to different LLC sets. When allocating physical pages, they choose the pages in different colors. Thus, the corresponding virtual pages are also mapped to different LLC sets.

For a few decades, PPMs have been widely used in mainstream operating systems, such as Linux, Windows, and FreeBSD [5], [6], [7]. They played an important role in improving LLC performance when LLC associativity was low. The increase of LLC associativity later reduces the dependence on PPMs. But PPMs are still equipped and enabled in most OSs, including the guest and host OSs/hypervisors in virtualized clouds. In view of the fact that way-based LLC partitioning in clouds reduces associativity and increases cache conflicts, it is natural to resort to PPMs and expect they can mitigate this issue.

However, virtualization makes existing PPMs completely ineffective in clouds [8]. The effectiveness of a PPM in mitigating LLC conflicts depends on its capability to properly "place" virtual pages onto the physical pages in different colors. On virtualized platforms, this is to place guest virtual pages (GVPs) into host physical pages (HPPs) in different colors[1]. However, none of the existing PPMs is managing such a cross-layer placement. For example, a PPM in a guest OS only manages the placement of GVPs in guest physical pages (GPPs) within a VM.

It is compelling to re-establish PPMs on virtualized platforms, because way-based LLC partitioning is increasingly supported by cloud system software and hardware processors, such as AMD, ARM, and PowerPC. A cross-layer page placement mechanism must be established not only under the conventional, non-nested

- *X. Shang, X. Ding, and C. Borcea are with Computer Science Department, New Jersey Institute of Technology, Newark, NJ 07102. E-mail: {xs225, xiaoning.ding, borcea}@njit.edu*
- *W. Jia is with Electrical, Computer, and Biomedical Engineering Department, The University of Rhode Island, Kingston, RI 02881. E-mail: weiwei.jia@uri.edu*
- *J. Shan is with Computer Science Department, Hofstra University, Hempstead, NY 11549. E-mail: Jianchen.Shan@hofstra.edu*

Manuscript received August 15, 2022.

1. The fixed mappings of HPPs to LLC sets are the indispensable leverage for a page placement mechanism to eventually correctly map virtual pages to LLC sets. Physical pages in VMs, including guest physical pages (GPPs) in non-nested virtualization, and GPPs and guest hypervisor physical pages (GHPPs) in nested virtualization, do not have fixed mappings to LLC sets.

virtualization configurations (NNVC), but also under the nested virtualization configurations (NVC), which become indispensable in many scenarios and is offered in most public clouds, including Azure, Google Cloud, and Amazon AWS.

To establish a cross-layer placement mechanism that can properly place GVPs onto HPPs in different colors, the key is a synergy between the PPMs at different system layers, including the guests, guest hypervisors (only under NVC), and the host. After all, the mapping between GVPs and HPPs ($f : GVP \mapsto HPP$) is the composition of the mappings managed by these PPMs. For NNVC, there are two mappings: 1) the *guest mapping* g between GVPs and GPPs ($g : GVP \mapsto GPP$) controlled by the guest PPM and 2) the *host mapping* h between GPPs and HPPs ($h : GPP \mapsto HPP$) controlled by the host PPM. Under NVC, they include three mappings: 1) the *guest mapping* g, 2) an *interposition mapping* $i : GPP \mapsto GHPP$, which is the mapping between GPPs and guest hypervisor physical pages (GHPPs), and is controlled by the guest hypervisor, and 3) the *host mapping* $h : GHPP \mapsto HPP$. In short, under NNVC, the fact $f = h \circ g$ determines that a desirable f should come from the synergy between h and g; under NVC, $f = h \circ i \circ g$ requires the synergy between all the three mappings h, i and g. Note that each mapping changes with the page allocation and deallocation at the corresponding layer. To keep synergy, mappings at other layers may need to change accordingly.

The establishment of a PPM across two layers has been studied and tested under NNVC in our previous work [8]. Under NNVC, for high portability, changes to guest OSs are usually avoided, including the changes to g. Thus, the synergy can only be achieved by adjusting h based on g. Note that adjusting h does not require the detection of g, i.e., monitoring all the allocations of GPPs to GVPs. Because the allocation of GPPs to GVPs is frequent, monitoring them will cause high overhead. Instead, to adjust h to g, we leverage a special feature of g: because the PPM in the guest functions as if it was on a physical machine, it extracts "virtual colors" from GPP addresses based on the set indexing of its virtual LLC, and places GVPs into GPPs in different "virtual colors". With the above feature of g, we only need to adjust h to ensure that the GPPs in different virtual colors can be mapped to the HPPs in different real colors. In this way, the adjustment of h is completely within the host and does not incur any interactions with guests. Thus, it has high portability and low overhead.

To reestablish PPMs under NVC, this paper designs XPLACE. The main challenge is that the solution must be implemented with low overhead at the host level (for high portability). However, compared to NNVC, the interposition of the guest hypervisor and i increases the design complexity (i.e., dealing with one extra layer of mapping) and implementation challenge (i.e., monitoring the changes in i and even g at the host level) for such as a solution. XPLACE takes advantage of the associative property of mapping composition to explore a viable approach that limits the solution within the host and keeps the overhead low. Specifically, it innovatively adjusts $h' = (h \circ i)$ to g, in light of the fact that $f = (h \circ i) \circ g = h' \circ g$. This enables XPLACE to leverage the aforementioned special feature of g to eliminate the overhead and implementation challenges in monitoring the page allocations in guests. However, the fact that h' is the composition of h and i still causes some challenges to XPLACE design. Because adjusting i is obviously not possible without changing guest hypervisors, adjusting h' can only be achieved by adjusting h based on i. This requires that the host must perform cross-layer monitoring to detect the dynamic changes of i, i.e., the allocations of GHPPs to GHPs in guest hypervisors. To detect changes of i, XPLACE uses the shadow page table mechanism, which is implemented to support page address translations under NVC. With the shadow page table mechanism, the VM page table is set to be "write-protected" by the host, and a change in i triggers a trap to the host. Thus, the host can take this opportunity to check and adjust h accordingly.

The paper makes the following contributions. First, to our knowledge, this is the first work that studies the cache conflict problem in the nested virtualization environment. Second, we have proposed XPLACE as an effective solution that can efficiently mitigate LLC conflict problem for nested virtualization; XPLACE addresses a few technical challenges, such as memory fragmentation. Finally, we have implemented XPLACE based on KVM in Linux kernel 5.3 and tested it with diverse applications in the nested virtualization environment. Our tests show XPLACE can significantly reduce cache conflicts and effectively improve application performance and system efficiency.

2 Background

2.1 Page Placement Mechanisms and Cache Conflicts

Page placement mechanisms (PPMs) were introduced when cache associativity was low (e.g., 1∼4) in early computer systems. Because hardware caches could not effectively absorb conflict misses due to the low associativity, page placement mechanisms were used as effective software mitigation. They were implemented and are still being used in main-stream system software, such as Linux, Windows, and FreeBSD [5], [6], [7].

A PPM reduces cache conflicts by improving the "placement" of virtual pages in physical pages, i.e., the allocation of physical pages to virtual pages. Leveraging the fixed mapping between physical pages and cache sets, it divides physical pages into disjoint groups. The pages in the same group are mapped to the same group of cache sets. For example, if each cache block is 64B, each group contains 64 cache sets because the page size is 4KB (4KB/64B=64). These cache sets are called a *cache color*. The number of cache colors is determined by the number of cache sets. For example, an LLC with 2048 cache sets has 32 cache colors if each cache color contains 64 cache sets (2048/64=32).

When allocating physical pages, a PPM needs to examine and consider the cache color that the physical page is mapped to. Thus, it uses the cache color indexes to "label" the pages and calls them *page colors*. The color of a page can be determined by examining its physical address based on the set indexing of the cache. For brevity, in the paper, the physical pages in the same color are called *conflicting pages*, and the physical pages in different colors are called *non-conflicting pages*.

For a set of virtual pages, to avoid the cache conflicts caused by visiting the data in these pages, the PPM allocates the physical pages in different colors to hold these virtual pages. In this way, the virtual pages are essentially mapped to different cache colors. Different page placement mechanisms use different policies to determine which virtual pages should be allocated with physical pages in different colors. For example, page coloring targets the workloads with sequential data access patterns and allocates non-conflicting pages to the virtual pages that are contiguous in virtual memory space. Bin-hopping targets repetitive data access patterns and allocates non-conflicting pages to the virtual pages that are consecutively accessed by each workload.

2.2 Two Types of Cache Partitioning Techniques

LLC partitioning mitigates the interference between workloads caused by sharing LLC space. There are two types of LLC partitioning techniques: 1) *set-based LLC partitioning*, where different LLC partitions contain different cache sets; and 2) *way-based LLC partitioning*, where different LLC partitions contain different cache ways.

Leveraging PPMs, set-based LLC partitioning assigns different workloads with the physical pages in different colors, such that these workloads can use different cache colors [9], [10]. Set-based cache partitioning has not been adopted in mainstream systems due to two issues: 1) cache and memory co-partitioning, i.e., large memory space must be reserved for a large cache partition, and 2) high overhead of recoloring virtual pages, i.e., replacing their physical pages from one color to another, since costly memory copying is involved in copying the page contents to new physical pages [10].

In clouds, the de facto practice is to use *way-based cache partitioning* with the hardware support built in processors, e.g., Intel CAT and AMD CAE. These supports allow the software to assign different LLC ways to different workloads [4]. For example, on Intel Xeon Gold 6138 processor, 20 cores share an 11-way LLC. If 11 workloads run on the processor, one in each VM, a VM can use a partition of 1 LLC way, similar to a direct mapped cache. Because the associativity of an LLC partition can be very low after way-based partitioning, LLC conflicts once again become a serious performance issue. Our experiments show that, without PPMs reducing LLC conflicts, the performance of cache-sensitive applications can be reduced by 51% with a 11-way partition; the performance degradation increases to 97% with a 1-way partition. Mitigating this issue relies on PPM to be effective.

The paper targets virtualized systems that use way-based LLC partitioning and studies how to reestablish PPMs to deal with the increased LLC conflicts on these systems. When PPMs become effective, they may be used to perform set-based LLC partitioning in VMs. We consider this to be beyond of the scope of the paper. First, set-based LLC partitioning has not been adopted in mainstream systems due to the issues mentioned above. Second, way-based LLC partitioning has become a de-facto solution. The scenarios that require the use of set-based LLC partitioning are rare.

2.3 Nested Virtualization and Its Memory Management

With virtualization technology, a hypervisor/host OS creates and manages virtual machines by emulating device hardware and allocating hardware resources dynamically. VMs have the same interfaces and functionalities of physical machines and can run unmodified operating systems (i.e., guest OSs) and applications.

Conventionally, hypervisors run directly on real hardware. Nested virtualization allows hypervisors (i.e., guest hypervisors) to run in VMs so as to host VMs inside VMs [11]. There are scenarios where the support of nested virtualization is indispensable. For example, Microsoft Windows 11 includes a type-1 hypervisor Hyper-V in it, which is to run Windows XP and the legacy applications relying on Windows XP; nested virtualization must be supported to run Windows 11 inside a VM [12]. There are also some scenarios where using guest hypervisors brings convenience. For example, using a guest hypervisor to contain multiple VMs can allow the migration of multiple VMs together to simplify management [11]; an extra layer of the hypervisor can help homogenize the diverse cloud infrastructures [13] or serve as a security monitor to isolate and protect VMs [14]; with nested virtualization, cloud users can also deploy their preferred hypervisors.

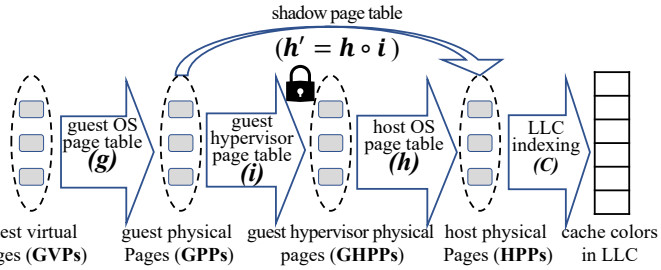


Fig. 1: Page mappings (g, i, h, and h') and LLC indexing c in nested virtualization.

Virtualization and nested virtualization allow unmodified operating systems and hypervisors to run inside virtual machines. Thus, memory is managed independently at each system layer using separate page tables. As shown in Figure 1, a guest OS uses its guest OS page table to maintain the mapping (g) from guest virtual pages (GVPs) to guest physical pages (GPPs); a guest hypervisor uses its guest hypervisor page table to maintain the mapping (i) from GPPs to guest hypervisor physical pages (GHPPs); and the host OS page table is used to maintain the mapping (h) from GHPPs to host physical pages (HPPs). The figure also shows the mapping of HPPs to LLC (c) controlled by hardware.

The mainstream hardware only supports the page translation in NNVC that uses two levels of page tables (i.e., two-dimensional page translation). However, the page translation under NVC requires the use of the information in three levels of page tables. To enable NNVC on the mainstream hardware, the host must merge two levels of page tables (usually the guest hypervisor page table and the host OS page table) into a shadow page table (SPT), in order to reduce the number of levels to 2. This is as illustrated in Figure 1 using the mapping h' between GPPs and HPPs. To merge the page tables, the host sets the guest hypervisor page tables to be "write-protected". Any update to a guest hypervisor page table (e.g., the allocation of a GHPP to a VM) will triggers a trap to the host. Thus, the host can take this opportunity to check the update and propagate the update to the shadow page table accordingly. When the host updates its host OS page table, it also propagate the changes to the SPT to keep consistency.

2.4 Ineffective Page Placement on Virtualized Platforms

As explained in Subsection 2.1, a PPM can reduce cache conflicts because it optimizes the mapping of virtual pages to cache colors. This can be achieved because 1) the PPM optimizes the mapping of the virtual pages to physical pages, and 2) the physical pages have fixed mappings to cache colors. On virtualized platforms, the virtual pages are the guest virtual pages (GVPs) at the guest level; each layer has its own type of physical pages, i.e., guest physical pages (GPPs), guest hypervisor physical pages (GHPPs), or host physical pages (HPPs). Because only HPPs have fixed mapping to cache colors, an effective PPM must optimize the mapping of GVPs to HPPs.

The mapping of GVPs to HPPs is the composition of the mappings at all different layers, including guest, guest hypervisor (only under NVC), and host. Currently, every PPM independently manages the mappings at its own layer. There is not a coordination

mechanism to optimize the composition mapping. Because GVPs cannot be properly "placed" on HPPs, LLC conflicts cannot be reduced with these PPMs. As shown in [8], LLC conflicts may increase by up to 44% on VMs under NNVC, relative to native systems. This problem can cause a 20% performance degradation under NNVC and may become even more pronounced under NVC.

CoPlace has been developed as a solution for NNVC [8]. This paper targets NVC and aims to develop a mechanism that coordinates the PPM in the host to the PPMs at upper layers and makes them function effectively as a cross-layer PPM. Note that we are not to improve existing page placement mechanisms which have already been proven to be very effective in reducing cache conflicts in a single layer after years of development and tuning. Instead, we aim to ensure that PPMs in multiple layers can coordinate to function correctly.

3 Possible Approaches

Since the issue is caused by managing the mapping from GVPs to HPPs and then to cache sets, it may be solved by replacing these mappings with a direct mapping from virtual pages to LLC sets (i.e., using guest physical addresses in LLC set indexing) [15]. However, the implementation of this virtual address indexing scheme requires changing hardware cache designs and complicates shared data handling.

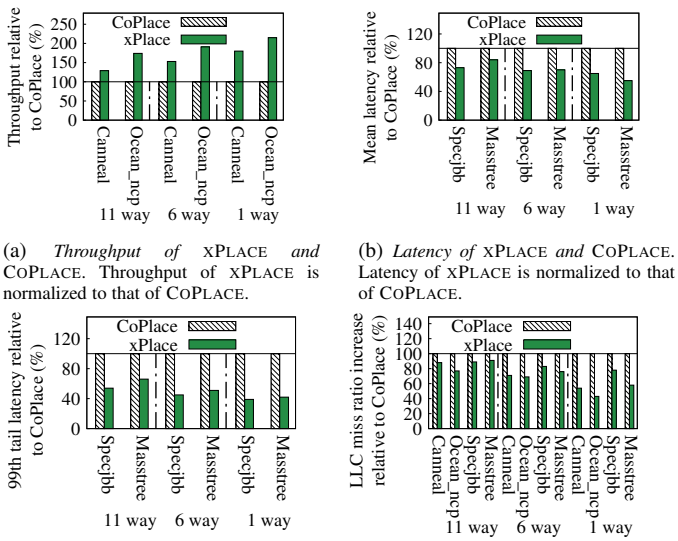
Another intuitive approach is to unify the mappings at different layers into one direct mapping from GVPs to HPPs. On the one hand, the nature of virtualization excludes such unification. On the other hand, this disallows the flexibility that different guests can adopt different page placement mechanisms that best benefit their workloads.

The causes that make existing PPMs ineffective are similar under NNVC and NVC. Another idea to solve this problem is to try CoPlace in the host under NVC. However, CoPlace was designed for non-nested virtualization which only incorporates the guest OS and the host OS. In the nested virtualization environment, the guest hypervisor introduces an extra layer of virtualization, making CoPlace unable to control the allocation of memory pages to a guest VM. Thus, CoPlace may not effectively reduce cache conflicts under NVC.

To show this, we compare the performance of CoPlace with xPlace, the approach proposed in this paper. We test the performance of two throughput oriented applications from `PARSEC` [16] benchmark suite, `canneal` and `ocean_ncp`, and two latency sensitive applications from `TailBench` [17], `Specjbb` and `Masstree`.

Please note that we test the performance of these applications with an application running in a dedicated VM on the server in this section. In Section 6, we extend the experiments to test the performance of the same application running in multiple VMs that are co-located on the server.

Figure 2a shows the throughputs of these applications with xPlace and CoPlace. We control the LLC space allocated to each application, and measure the performance with three different LLC space sizes (1-way, 6-way, and 11-way). On average, xPlace outperforms CoPlace by 51% for 11 way LLC allocation, 72% for 6 way LLC allocation, and 97% for 1 way LLC allocation. This matches the LLC miss ratio increase as shown in Figure 2d. In comparison to CoPlace, xPlace offers 14%, 25%, and 42% lower LLC miss ratio increase with 11 way LLC allocation, 6 way LLC allocation, and 1 way LLC allocation,



(a) *Throughput of* xPlace *and* CoPlace. Throughput of xPlace is normalized to that of CoPlace.

(b) *Latency of* xPlace *and* CoPlace. Latency of xPlace is normalized to that of CoPlace.

(c) *99 percentile tail latency of* xPlace *and* CoPlace. Latency of xPlace is normalized to that of CoPlace.

(d) *LLC miss ratio increase of* xPlace *and* CoPlace. LLC miss ratio increase of xPlace is normalized to that of CoPlace.

Fig. 2: CoPlace Is Ineffective under NVC.

respectively, for the throughput oriented applications. We notice that xPlace is more effective in improving throughput when fewer LLC ways are used, as cache conflicts are more serious.

For latency sensitive applications, Figure 2b and Figure 2c show their mean latencies and 99th tail latencies with xPlace and CoPlace and under three different LLC space sizes, 1-way, 6-way, and 11-way, respectively. Compared to CoPlace, xPlace reduces mean latency by 31% and 99th tail latency by 51% on average. Similar to what has observed with throughput oriented applications, xPlace demonstrated similar trends with latency sensitive applications. It is more effective in reducing latencies when fewer cache ways are used, and the latency reduction correlates well with the reduction of LLC misses shown in Figure 2d.

4 Problem Overview and Challenges

This section analyzes and then formally defines the problem of reestablishing page placement mechanisms on virtualized platforms. Then it explains the challenges in establishing a cross-layer page placement mechanism under NVC.

4.1 Problem Statement and Analysis

On the existing hardware and the layered software architecture for virtualization, for a workload, its virtual pages (V) are mapped to HPPs by the composition mapping f of a few component mappings, i.e., $f = h \circ g$ under NNVC and $f = h \circ i \circ g$ under NVC. These component mappings change with page allocations and deallocations. The HPPs are further mapped to LLC cache colors by mapping c. All these mappings are as shown in Figure 1 under NVC. Without loss of generality, we assume that g, i, and h are injective, and c is non-injective.

The problem of reestablishing page placement mechanisms on a virtualized platform is to ***make $c \circ f$ an injective mapping on any P_v, which is a subdomain of V containing a group of GVPs that are selected by a PPM to reduce LLC conflicts***. The injectivity of $c \circ f$ ensures that different GVPs in P_v are mapped to different LLC colors. Without loss of generality, we assume that a PPM is equipped in the guest. Since the workload runs in the guest, the guest PPM is the one that decides which GVPs should be included

in the same subdomain P_v to avoid conflicts. We also assume that it controls the size of P_v for the number of GVPs in P_v not exceeding the count of LLC colors, such that all GVPs in P_v can be mapped to different LLC colors.

Since f is a composition of its component mappings and c is a fixed mapping built in hardware, to make $c \circ f$ injective on P_v, a solution to the problem must effectively controls f by managing its component mappings. In the paper, we use ***synergy*** to refer to the capability of the component mappings to build up a f that makes $c \circ f$ injective on P_v.

The problem should be considered under a portability constraint: for the high portability of a solution, changes should be limited within the host OS; and changes to guest OSs or guest hypervisors should be avoided. Under this constraint, the synergy between the component mappings of f can only be achieved by adjusting h to respond to the changes in other component mappings; and the problem becomes ***how to dynamically adjust*** h ***upon the changes of other component mappings, so as to keep*** $c \circ f$ ***an injective mapping on any*** P_v.

Solving this problem relies heavily on the information maintained at the guest layer and guest hypervisor layer. It needs to be aware of each subdomain P_v formed in the guest, monitor every change of the other component mappings (g and i), and adjust h accordingly to prevent $c \circ f$ becoming non-injective on P_v. However, with the semantic gap and layered architecture formed by virtualization, memory management and dynamic changes in page mappings in one layer are transparent to other layers. To monitor and obtain all the information above, both implementation challenges and overhead are daunting.

4.2 Challenges on Nested Virtualization Systems

The challenges in solving the problem under NNVC first lie in the complexity of the problem, most of which is introduced by guest hypervisors. Compared to the 2-layer architecture under NNVC, this extra layer not only introduces one more mapping (i) that must be dealt with in the solution, but also adds an obstacle that impedes the solution implemented in the host from getting enough information of the guest. The challenges also lie in the fact that the solution for NNVC is ineffective under NVC, as we show in Section 3, and cannot be easily extended to address these challenges, as we will explain below.

CoPlace addresses the challenge under NNVC by first reformulating the problem into a "stronger" problem, which "implies" the original problem and is irrelevant to the mapping g in the guest. The reformulation leverages a special feature of g formed by the PPM in each guest: because the PPM performs page placement as if it was on a physical machine, the GVPs in P_v are mapped to different virtual cache colors in the virtual LLC. Let c' be the mapping between GPPs and virtual LLC colors. c' ***is injective on any*** $g(P_v)$.

Instead of solving the original problem, which requires the detection of every change in g, CoPlace tries to solve a "stronger" problem: how to ensure that the GPPs in different virtual colors are backed by the HPPs in different real colors. Under the portability constraint, this stronger problem can be further reformulated as ***how to manage*** h***, such that a fixed and injective mapping*** t ***between the virtual colors and the real colors of all GPPs can be formed and maintained, i.e., managing*** h***, such that*** $\forall GPP, c \circ h = t \circ c'$***, where*** t ***is injective.*** The special feature above determines that a solution of this stronger problem is also a solution of the original problem.

The reformulations turn a cross-layer synergy problem between g and h into a single-layer injectivitiy problem (i.e., maintaining a t in the host). CoPlace solves this problem by initiating a table t when a VM is created, and referring to t for the colors of the HPPs when it selects HPPs to back GPPs. This solution relies only on the information available in the host. Though virtual colors is the information of the guest, it is also available to the host as a part of VM interface: the virtual LLC is created and configured by the host; and the GPPs are managed by the host and provided to the VM as its physical memory space.

The effectiveness of the CoPlace approach is built upon the 2-layer guest-host architecture: with the host being the layer immediately under the guest, it can utilize the information shared as a part of the VM interface; with the host being the only layer between the guest and hardware, it can control the mapping between virtual colors and real colors. Under NVC, the interposition of the guest hypervisor layer makes the approach ineffective. It also creates a substantial obstacle for the host to gain the above capabilities. Extending this approach must overcome this obstacle.

5 xPLACE Design

5.1 Basic Idea and Overview

Solving the problem under NVC requires a synergy between three page mappings (i.e., g, i, and h) that are managed at three different system layers. Our solution is based on the feature of the multi-layered software architecture and the associative property of mapping composition. We first take the guest hypervisor and the host as a "composite host", and then consider the problem of adapting this "composite host" to create a synergy between it and its guest. Specifically, we take $f = h \circ i \circ g$ as $f = h' \circ g$ where $h' = h \circ i$, and then consider how to adjust h' dynamically. We take the guest hypervisor and the host as a "composite host" for two reasons: 1) CoPLACE takes the guest hypervisor and the guest as a "composite guest", and proves to be an ineffective approach (§3); and 2) the eventual solution will be built in the host, and the host has direct interactions with the guest hypervisor. Consider them as a "composite host" help leverage the existing cross-layer mechanisms, such as shadow page table, as we will explain below.

Under this 2-layer conceptual architecture of guest and "composite host", after the problem reformulations that are similar to those in CoPLACE (§4.2), the problem becomes ***how to manage*** h'***, such that a fixed and injective mapping*** t ***between the virtual colors of GPPs and the real colors of the HPPs backing these GPPs can be formed and maintained, i.e., managing*** h'***, such that*** $\forall GPP, c \circ h' = t \circ c'$***, where*** t ***is injective.*** The reformulations are also to minimize the reliance of xPLACE on the information in the guest.

Since h' is a composition of mapping i and mapping h, managing h' needs to 1) monitor and handle the changes of i, and 2) control the changes to h. The monitoring and handling of the changes of i are implemented in the host page placement mechanism. Without changing the existing layered structure, it looks to be a challenging issue to monitor and handle the changes of i, because the solution is in the host, and i is managed in the guest hypervisor. We address these issues by leveraging the shadow page table (SPT) mechanism. SPT is an "already-to-use" mechanism for the bookkeeping of h' in the host. With SPT, any change to i (e.g., the allocation of a GHPP to back a GPP) will incur a trap to the host. Taking this opportunity, the host can examine this change by checking the corresponding GPP (for a

virtual color), its GHPP, and then the HPP (for the real color). If this change may make t become non-injective, the host page placement mechanism adjusts h by replacing the HPP backing the GHPP with another HPP with a desired color, such that t can continue to be injective. This will be explained in §5.2.

To control the changes to h, we enhance the memory management components that may change the page mappings in the host. These components include *buddy allocator*, which allocates host physical pages with low memory fragmentation, *memory deduplication*, which merges physical pages with identical contents to reduce physical memory consumption, and *memory ballooning*, which adjusts the physical memory space available to the workloads in each VM. These components need to refer to shadow page tables for the virtual colors of GPPs and to coordinate the requirement of keep t injective with their original design goals. We introduce the enhancements in §5.3 and §5.4.

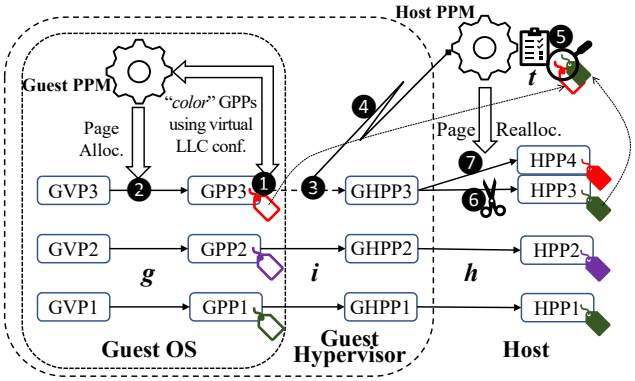


Fig. 3: XPLACE PPM Creates Synergy with Guest PPM

5.2 Host Page Placement Mechanism in XPLACE

In XPLACE, the host page placement mechanism monitors and handles the changes to i so as to maintain the injectivity of t. It starts with setting up the mapping t. When a VM is created and the virtual LLC configuration in the VM is set up based on the real LLC configuration, XPLACE assigns a unique real color to each virtual color. (The number of cache sets and cache colors in the virtual LLC is usually set to be the same as the real LLC.) It maintains the color assignments in a table. We also refer to this table as t.

After the VM is launched, the host PPM keeps monitoring and handling the changes in i. We use Figure 3 to illustrate the steps. To better explain the idea of XPLACE as a whole, the figure also includes the page management in the guest. As shown with the page mapping g inside the dotted line rectangle area representing the guest OS, the PPM in the guest OS selects and allocates GHPs in different virtual colors to GVPs. Specifically, the guest PPM determines the virtual color of a GHP using its guest physical page addresses and the configuration of the virtual LLC (❶). The virtual color is illustrated using a hollow tag icon. Using virtual colors, the guest PPM performs page placement/allocation in the same way that it would on a physical machine (❷).

With the shadow page table mechanism, the host PPM in XPLACE monitors every change in the mapping i between GPPs and GHPPs. For example, when the guest hypervisor is allocating a GHPP (e.g., the allocation of GHPP3 shown with ❸), it needs to change the write protected page table in the guest hypervisor. This incurs a trap to the host (❹). Note that the monitoring incurs minimal overhead. It leverages the trap caused by the original shadow page table mechanism in nested virtualization and does not incur extra traps.

The host PPM in XPLACE detects whether such change of i may destroy the injectivity of t, i.e., the synergy. This is as shown with ❺. For the simplicity of the explanation, we assume that an identity mapping is selected and used as t in Figure 3, i.e., real colors matching virtual numbers. The trap contains the address of GPP, from which the host PPM can extract the virtual color, and the address of the GHPP, with which the host PPM can look up the host OS page table to locate the HPP backing the GHPP and then determine the real color. Thus, it can look up the table t using the virtual color (e.g., the virtual color of GPP3 in the figure), finds the real color assigned to the virtual color (e.g., red), and compares it against the real color (e.g., the green color of HPP3, illustrated using a solid tag icon).

If the two real colors do not match, XPLACE determines that the injectivity/synergy is destroyed; it replaces the HPP to restore the synergy. It frees the HPP (❻), finds another HPP in a matching color (i.e., GPP4 in red), and allocates the new HPP to back the GHPP (❼). Note that in this page re-allocation process, no memory copying is required, because both HPPs (i.e., HPP3 and HPP4) are free pages containing no valid data. The only overhead for restoring the synergy is looking for a HPP and changing the corresponding pointers (free list and page tables) to finish the allocation.

Also note that in the above example we assume that a HPP has been allocated to back the GHHP when the GHHP is being allocated in the guest hypervisor. In the case that a HPP has not been allocated, XPLACE also looks up table t with the virtual color of the GHP, and allocates a HPP in the corresponding color.

5.3 Improving Buddy Memory Allocator

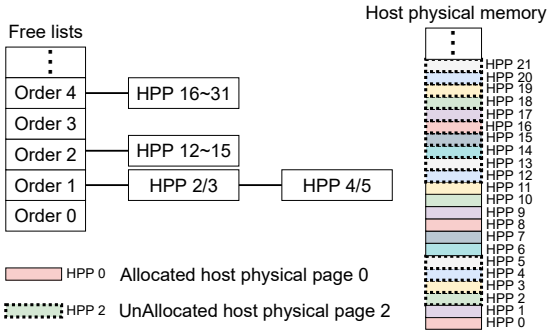


Fig. 4: Linux buddy allocator. There are eight cache colors (i.e., cache colors of HPP 0∼7) in this example. To realize XPLACE, host memory fragmentations may be increased based on the current buddy allocator mechanisms. For instance, when four host physical pages (HPPs) with cache colors of HPP 2∼5 are requested, a larger memory block (i.e., HPP 16∼31 in order-4) has to be split. This is because current buddy allocator searches from order-2 and the memory block of HPP 12∼15 does not satisfy the cache color requirement.

For a memory allocator, reducing memory fragmentation is one of the most important design goals. When free pages are separated by allocated pages into small memory blocks, allocating large chunks of physically contiguous memory becomes difficult or may fail. Binary buddy memory allocator is now widely used in operating systems and hypervisors, including Linux and KVM, for its efficiency and capability to deal with memory fragmentation [18], [19], [20], [21]. The buddy allocator groups free memory pages into blocks. Each block can contain 2^x contiguous free pages and is

aligned to $2^x \times 4KB$, where the non-negative number x is the *order* of the block. Order 0 blocks are individual free pages. As shown in Figure 4, blocks of the same size are organized on the same list; the order-1 list has two blocks, one containing host physical pages #2 and #3 ("HPP 2/3"), and the other containing #4 and #5 ("HPP 4/5"). To minimize fragmentation, on a memory allocation request, the allocator finds a free memory block of the smallest size that can satisfy the request, e.g., a block on the order-2 list for a request of 3 pages, and preserves free memory blocks with larger sizes. When a memory block is freed, the buddy allocator tries to combine it with other free blocks on the lists to form a larger free memory block.

As explained in the previous subsection, to reduce LLC conflicts, XPLACE needs to check the colors of the guest physical addresses and allocate the host physical pages in the corresponding colors determined by the mapping. It is possible that the requested memory blocks do not have the free pages in the required color. To satisfy the requirements on page colors, a larger memory block must be split into pieces with one piece used to satisfy the request and other pieces moved to low order lists. This increases memory fragmentations.

For instance, in Figure 4, when the allocator is requested to allocate four HPPs with the colors matching those of HPP 2∼ HPP 5, it first checks the order-2 list. Though there are 4-page blocks (e.g., HPP 12∼15) on the list, they cannot satisfy the requirements on colors. In this case, an order-4 block (i.e., HPP 16∼31) must be split.

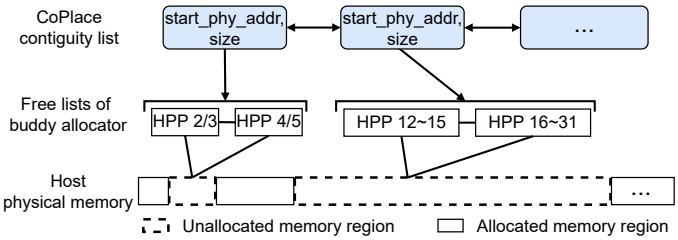


Fig. 5: XPLACE contiguity list. The contiguity list is sorted based on the size of the free memory region. Upon page fault, XPLACE searches the contiguity list to find the smallest free memory region with designated cache colors for memory allocations.

The problem is caused by the requirements of buddy allocator on block sizes and alignment (i.e., 2^x pages). Some pages (e.g., HPPs 2∼5) are contiguous; but they cannot be organized into a block if they cannot meet both requirements. To solve this problem, XPLACE uses a contiguity list to track the contiguity of the host free memory regions on top of the buddy allocator and allocates HPPs with designated cache colors based on the contiguity list. Figure 5 illustrates the contiguity list. It includes the host free memory regions. Each free memory region is described with the starting host physical address and the size. The list is sorted based on the size of the free memory region. Upon page fault, XPLACE searches the contiguity list to find the smallest free memory region that can satisfy the memory allocation requirement. Then, those requested HPPs are allocated from the corresponding free list of the buddy allocator. This can mitigate host memory fragmentations. For instance, in Figure 4, when XPLACE is requested to allocate four HPPs with cache colors of HPP 2∼5, it allocates the two memory blocks in order-1 after searching the contiguity list. To yield better performance, we leverage the red black tree in Linux to realize the contiguity list.

5.4 Improving Memory Deduplication and Ballooning

For memory deduplication (e.g., Linux same page merging in KVM [22]), and memory ballooning [23], the COPLACE enhancements on their policies can be used directly; but the implementations must be changed to look up shadow page tables in order to obtain the virtual LLC colors of GPPs.

We have not seen any discussions on how memory deduplication or memory ballooning should be performed on nested virtualization environments. XPLACE assumes that memory deduplication is enabled only in the host OS and not in guest hypervisors, since this is the most efficient choice. Because the host OS can identify and merge the identical pages in different VMs and different guest hypervisors, performing memory deduplication inside each guest hypervisor is unnecessary and causes extra overhead. XPLACE suggests VMs install ballooning drivers from the host OS (not the guest hypervisor) for the highest efficiency, because a set of issues can be avoided or mitigated, such as double swapping, and memory thrashing that may be possibly caused by uncoordinated two layers of memory ballooning.

6 Evaluation

To evaluate XPLACE, we implemented a prototype on Linux/KVM 5.3, based on CoPlace [8], with 110 lines of source code added or modified in the host memory manager. Specifically, we made changes to the code that handles the shadow page table and the design of Linux per CPU page (PCP) lists [24]. The PCP lists maintain a pool of free memory pages for each core, enabling fast allocation and de-allocation of pages. The pool is sized to be reasonably large and when the watermark is low, it is refilled with a large batch of pages from the main memory pool managed by the buddy allocator. For high efficiency, we reorganized the data structures of the PCP lists to manage pages based on colors. This allows XPLACE to quickly obtain the free pages it needs in certain colors from the PCP lists, rather than going through a slower execution path and using the buddy allocator.

We conducted a thorough evaluation of XPLACE using the prototype implementation and a diverse set of workloads. Especially, to assess its effectiveness and advantage in nested-virtualized clouds, we compared it with COPLACE in terms of the capability of reducing cache conflicts and improving application performance in the nested virtualization environment.

6.1 Experimental Settings

Our experiments used a Hewlett Packard Enterprise ProLiant DL580 Gen10 server. The server is equipped with four Intel Xeon Gold 6138 processors, 256GB memory, two 2TB HDDs, and two 2TB SSDs. The processors support VMCS shadowing [25] for better nested virtualization performance. Each processor has 20 cores, which share a 11-way 27.5MiB LLC. Each core has a 32KiB L1d cache, a 32KiB L1i cache, and a 16-way 1MiB L2 cache. We used KVM/QEMU to build the nested virtualization environment. The virtual machine has 16 vCPUs and 8GiB memory. Host OS, guest hypervisor, and guest OS are all Ubuntu 18.04 with kernel updated to 5.3. Our evaluation was conducted by running benchmarks in a single VM or 2 VMs. When a single VM is used for experiments, the guest OS uses bin-hopping page placement mechanism. When multiple VMs are used for experiments, the guest OSs use different page placement mechanisms: page coloring and bin-hopping[2]. For bin hopping, we use the default page placement mechanism

App.	Workload description
Img-dnn	Handwriting recognition based on OpenCV [27].
Sphinx	Speech recognition like Apple Siri [28].
Moses	Real time translation like Google translate [29].
Xapian	Search engine used in websites and S/W frameworks [30].
Masstree	In memory K/V store with 50% GET and 50% PUT [31].
Specjbb	Industry-standard JAVA middleware benchmark.
Silo	In-memory transactional database with TPCC [32].
Shore	On-disk transactional database with TPCC [33].
PARSEC	Seven benchmarks from PARSEC benchmark suite.
SPLASH2X	Five benchmarks from SPLASH2X benchmark suite.

TABLE 1: Programs and workloads used in experiments.

in Linux; for page coloring, we implement the FreeBSD's page coloring mechanisms [26] into Linux.

We evaluated XPLACE using a large set of workloads (listed in Table 1). They are representative workloads in various application domains, including database server, web server, key-value store, search engine, and AI training. We categorize them into two groups: throughput-oriented workloads and latency-sensitive workloads. We tested 12 throughput-oriented workloads from PARSEC and SPLASH2X benchmark suites and 8 latency-sensitive workloads from TailBench. We collected throughputs of throughput-oriented workloads, and mean and tail latencies of latency-sensitive workloads. To understand the performance of XPLACE and COPLACE, we collected and compared the increase in the LLC miss ratio when a workload is moved to the nested-virtualized environment facilitated with XPLACE and COPLACE, using the LLC miss ratio of each workload in the bare-metal environment as the baseline.

To show the impact of LLC partitioning on cache conflicts, we used two configurations for measurements: (1) 11-way LLC allocation: each workload is allocated with the whole LLC space (i.e., 11-way), (2) 1-way LLC allocation: each workload is allocated with 1-way LLC space. We leverage Intel Cache Allocation Technology (CAT) to allocate LLC partitions.

We executed each workload immediately after booting the VM. By starting with a clean slate, we were able to achieve stable performance across multiple runs. For each setting (LLC space size and page placement policy), we executed the workload using XPLACE and COPLACE, respectively, and compared their performance. To facilitate the presentation and comparison of results, we normalized the throughput, latency, and LLC miss ratio measurements obtained using XPLACE against the corresponding values obtained using COPLACE.

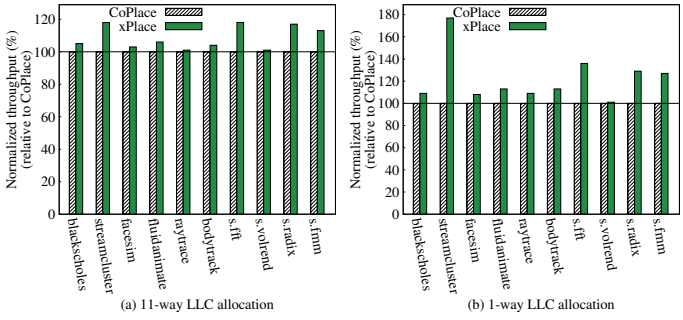


Fig. 6: ***Throughput of* XPLACE *and* COPLACE *when the throughput-oriented workloads are tested.* XPLACE's throughput is normalized to that of COPLACE.**

2. With this setting, we want to test whether XPLACE allows different guests to use different page placement mechanisms that best fit their workloads.

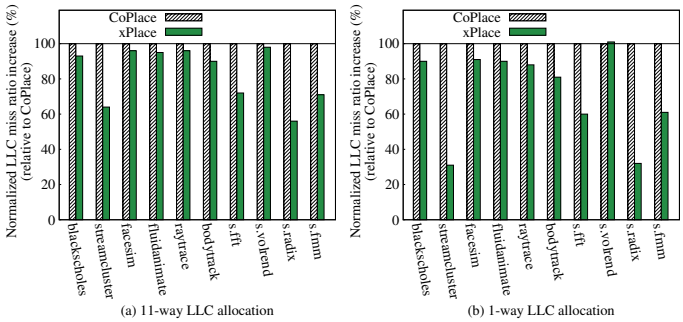


Fig. 7: ***LLC miss ratio increase of* XPLACE *and* COPLACE *when throughput oriented workloads are tested.* XPLACE's LLC miss ratio increase is normalized to that of COPLACE.**

6.2 Experiments with Throughput-Oriented Workloads

Figure 6 shows the measurements of each throughput-oriented workload when XPLACE and COPLACE are tested under 1-way LLC allocation and 11-way LLC allocation, respectively. On average, XPLACE outperforms COPLACE by 22% for 1-way LLC allocation and 8% for 11-way LLC allocation. It's worth noting that 1-way LLC allocation significantly reduces the associativity of the LLC space available to the workload, which increases the difficulty to reduce cache conflicts and may cause serious cache conflicts. However, XPLACE offers 14% more throughput improvement when LLC is allocated with 1-way, which suggests that XPLACE is effective in reducing cache conflicts and improving application performance, even in a nested virtualization environment where the associativity of LLC allocation is low.

To illustrate the effectiveness of XPLACE in reducing cache conflicts, the LLC miss ratio is profiled when a workload is executed in each evaluated system. The results are presented in Figure 7, which shows that XPLACE causes less increase in cache conflicts than COPLACE does. Figure 7 (a) shows that when the LLC allocation is 11-way, XPLACE reduces the LLC misses ratio increase by 16% on average, compared to COPLACE. Figure 7 (b) shows XPLACE incurs 28% lower LLC miss ratio increase on average, compared to COPLACE, when the LLC is allocated with 1-way partition. Among all the workloads, compared to COPLACE, XPLACE achieved the highest reduction in cache conflicts with streamcluster, which is aligned with streamcluster's largest throughput improvement as shown in Figure 6. Streamcluster groups a stream of input points into different clusters by finding a predetermined number of medians so that each point is assigned to its closest center. As a memory-intensive program, it incurs much higher LLC conflicts when the associativity of LLC partition decreases since the same working set of streamcluster needs to fit into a much smaller LLC space (i.e., 1-way LLC space vs. 11-way LLC space).

6.3 Experiments with Latency-Sensitive Workloads

More latency-sensitive (e.g., Spark) applications are being deployed in clouds now. To evaluate how XPLACE can benefit them, we evaluated typical latency-sensitive workloads and report their mean and tail latencies in Figure 8. Figure 8 (a) and (b) show that XPLACE reduces the mean latency by 6% for 11-way LLC allocation and 25% for 1-way LLC allocation on average, compared to COPLACE. Figure 8 (c) and (d) show that, in comparison to COPLACE, XPLACE provides 21% and 41% lower 99th tail latency on average for 11-way and 1-way LLC allocation, respectively. Our experiments show that tail latencies

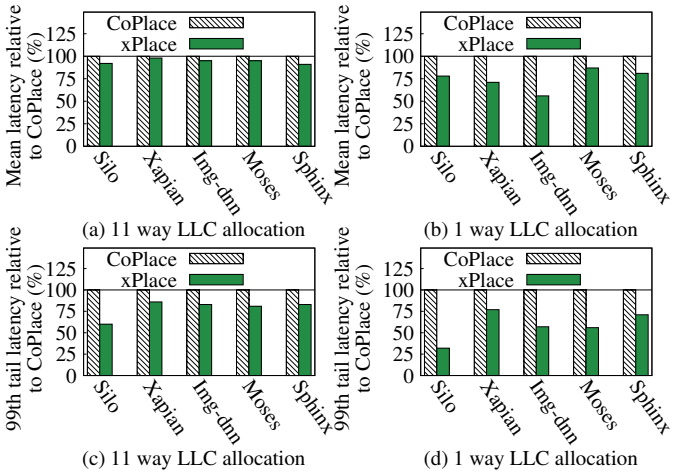


Fig. 8: ***Latencies of* xPLACE *and* CoPLACE *when latency sensitive workloads are tested.* xPLACE's mean and tail latencies are relative to those of CoPLACE.**

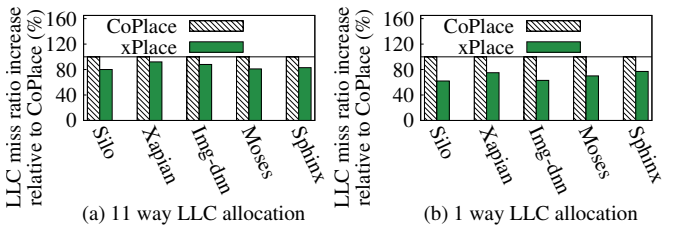


Fig. 9: ***LLC miss ratio increase of* xPLACE *and* CoPLACE *when latency sensitive workloads are tested.* xPLACE's LLC miss ratio increase is normalized to that of CoPLACE.**

are more vulnerable to LLC misses. For example, with xPLACE, `Img-dnn` shows 24% lower mean latency and 30% lower 99th tail latency relative to those with CoPLACE. `Img-dnn` is a handwriting application that leverages a deep neural network-based auto-encoder combined with softmax regression to identify handwriting characters. The input data are some samples from the MNIST dataset. This shows xPLACE can greatly benefit AI workloads if they are cache sensitive.

To pinpoint the performance advantage of xPLACE compared to CoPLACE, we also profile the LLC miss ratio of the latency-sensitive workloads when they are tested with xPLACE and CoPLACE, respectively. We show the profiling results in Figure 9. On average, xPLACE reduces the LLC miss ratio increase by 15% for 11-way LLC allocation and 31% for 1-way LLC allocation compared to CoPLACE.

xPLACE significantly reduces LLC miss ratio increase compared to CoPLACE due to two key reasons. Firstly, xPLACE's design incorporates page placement mechanisms that are tailored for nested virtualization environments, while CoPLACE's design does not. Nested virtualization adds an extra layer of guest hypervisor, making the semantic gap of virtualization more complex and exacerbating the cache conflict problem. CoPLACE's design only caters to non-nested virtualization environments and cannot be easily extended for nested virtualization. xPLACE, on the other hand, leverages the shadow page table between the guest hypervisor and host hypervisor to allocate non-conflicting host physical pages for non-conflicting guest physical pages that are used to execute workloads running on the guest OS. Secondly, the existing buddy memory allocator may increase memory fragmentation when allocating conflicting/non-conflicting host physical pages

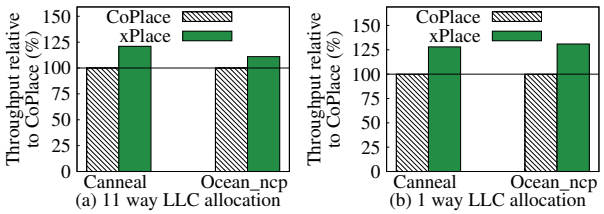


Fig. 10: ***Throughput of* xPLACE *and* CoPLACE *when two VMs use different page placement mechanisms*. Throughput of xPLACE is normalized to that of CoPLACE.**

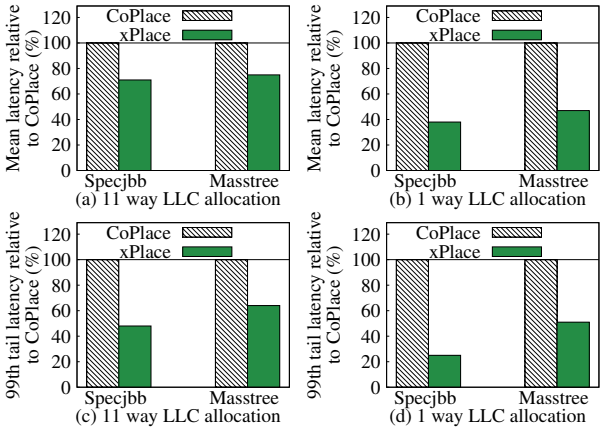


Fig. 11: ***Latency of* xPLACE *and* CoPLACE *when two VMs use different page placement mechanisms*. Latency of xPLACE is normalized to that of CoPLACE.**

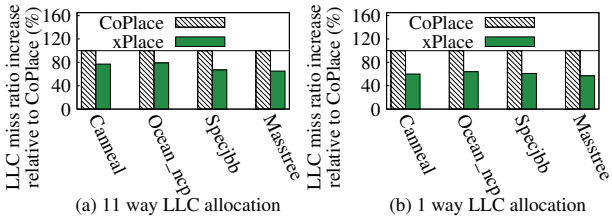


Fig. 12: ***LLC miss ratio increase of* xPLACE *and* CoPLACE *when two VMs use different page placement mechanisms*. xPLACE's LLC miss ratio increase is normalized to that of CoPLACE.**

for corresponding guest physical pages, potentially degrading application performance. xPLACE's design addresses this challenge by implementing a buddy memory allocator that minimizes memory fragmentation, thereby enhancing the performance of cloud workloads.

6.4 Flexibility

xPLACE provides high flexibility, which allows each VM to select a page placement mechanism that fits best its workload. To demonstrate this flexibility, we create two VMs (denoted as VM1 and VM2) on the same guest hypervisor. VM1 uses a bin-hopping policy, and VM2 uses a page coloring policy. We test xPLACE using two throughput-oriented workloads (`canneal` in VM1 and `ocean_ncp` in VM2) and two latency-sensitive workloads (`Specjbb` on VM1 and `Masstree` on VM2).

As shown in Figure 10 and Figure 11, xPLACE can effectively support different page placement mechanisms used in different VMs. Compared to CoPLACE, xPLACE can increase the throughput by 16% with a 11-way LLC allocation and 29% with a 1-way LLC allocation. xPLACE can decrease the mean latency by 27% and the 99th tail latency by 44% when the LLC allocation is 11-way. The improvement is higher when the VMs use a 1-

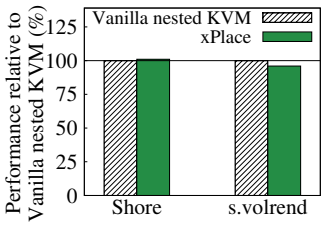


Fig. 13: xPLACE's *overhead*. Performance of xPLACE is normalized to that of Vanilla nested KVM.

way cache partition, as there are more cache conflicts to be reduced. Figure 12 confirms that the improvements in throughputs and latencies are from xPLACE reducing LLC misses. CoPLACE cannot effectively reestablish page placement mechanisms for nested virtualization. The LLC miss ratios are much higher than those on bare metal. With xPLACE, page placement mechanisms reduce conflict misses. The LLC miss ratios are substantially lower than those with CoPLACE.

6.5 Overhead

xPLACE incurs minimal overhead. It monitors and responds to the page allocations to VMs. The monitoring does not incur any overhead since it only leverages the existing traps to the host. The only significant overhead is incurred when replacing the HPPs used to back GHPPs. This overhead mainly includes looking for a new HPP with a desired color, changing the page mapping, and putting the replaced HPP back on a free page list. However, the overhead is low because these operations mainly update "pointers". Replacing HPPs in XPlace does not involve copying or resetting page contents, because HPPs are free pages and contain no valid data.

To test the overhead incurred by xPLACE, we used benchmarks that are not cache-sensitive. As there is almost no space for xPLACE to improve their performance, the overhead of xPLACE is determined by measuring the slowdown between their performance with xPLACE and their performance with vanilla KVM. Specifically, we selected `Shore` [33], which is a transactional database driven by a TPC-C workload, and `Volrend`, that renders a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing. We run each benchmark in a VM multiple times. To capture the scenario where xPLACE may incur the highest overhead, we relaunch the VM after each execution without relaunching the guest hypervisor. In this scenario, the GHPPs allocated to the VM are backed by the HPPs that are usually not in desired colors. Thus, the aforementioned cost of replacing the HPPs used to back GHPPs must be paid. As Figure 13 shows, the performance of `Shore` and `Volrend` is degraded by about 2% with xPLACE, compared to Vanilla nested KVM. This confirms that the overhead of xPLACE is very low.

7 Related Work

Reducing cache conflicts in virtualized clouds. Existing techniques [5], [34], [35] for reducing cache conflicts include page coloring [34], bin-hopping [5] and their variants. Page coloring [34] allocates continuous virtual pages mapping to different cache colors. It mitigates cache conflicts because sequential virtual pages do not conflict with each other in the cache. However, page coloring may lead to excessive inter-address-space contention because it may map commonly used virtual addresses (e.g., stack) of different processes to the same cache color. PID hashing (a match of cache color bits exclusive-ored with the process's identifier) is used to mitigate the problem.

Bin hopping [5] places successively allocated physical pages in successive bins (a bin includes all the pages that have the same cache color), irrespective of their virtual addresses. It reduces cache conflicts because it sequentially distributes the mapped physical pages from an address space across different bins. It exploits temporal locality because the physical pages it maps close in time tend to be placed in different bins. Best bin selects a page frame from the bin with the fewest previously allocated and most available page frames. Hierarchical bin is a tree-based variant of best bin that executes in logarithmic time and produces cache size-independent placement improvement. Previous work [35] also shows that the random placement puts many pages in the same cache bins; and this competition is undesirable since it can cause more cache conflicts.

Existing techniques for mitigating cache conflicts may be ineffective in virtualized clouds. Our previous work [8] (i.e., CoPLACE) designed for the non-nested virtualization environment shows existing approaches for reducing cache conflicts are ineffective in virtualized clouds due to the semantic gap between guest OS and host OS. The semantic gap becomes more complex and makes CoPLACE inefficient in the nested virtualization environment. This work proposes xPLACE as an effective solution to further reduce cache conflicts and improve application performance in the nested virtualized clouds.

Reducing cache interference with partitioning. Proposals on cache partitioning mainly include software and hardware approaches. For software approaches, many works rely on page coloring to partition the cache space [9], [36]. The basic idea is to allocate cache regions with specific cache colors to one application such that its cache space is isolated. [37] propose vLLC (virtual last level cache) and vColoring techniques to partition cache space for VMs. vLLC is used for coloring-aware guest OSs. It lets host OS notify guest OS the information about its allocated cache partition, e.g., cache capacity, number of cache sets, number of cache colors; then, host OS controls the mappings between GPAs (guest physical address) to HPAs (host physical address) based on its allocated cache capacity and colors. For vLLC, host OS allocates cache partition for the guest OS, making tasks running in the guest OS cannot control its cache colors. vColoring extends the idea of vLLC and is used for coloring-unaware guest OSs. It proposes two sets of cache colors: default cache colors and extra cache colors. The default cache colors are used when GPAs map to HPAs by default. This is done by the hypervisor. The extra cache colors are used when tasks in VMs explicitly request new cache colors except the default set of cache colors. This is done by migrating already allocated and present pages of the tasks to new host physical pages mapping to the new cache colors. This may incur high overhead due to VM exits caused by hypercalls and page migration. More importantly, vLLC and vColoring may not effectively mitigate cache conflicts for tasks in VMs because host physical addresses are agnostic to these tasks and cache partition is essentially allocated and controlled by hypervisor. Specifically, hypervisor just allocates VMs multiple cache colors (can be regarded as cache partitions for VMs) and notify VMs, and then VMs use the cache partitions without fine-grained controlling of the mappings between host physical addresses and cache colors. This work is not designed for the nested virtualization environment.

For hardware approaches, existing hardware extensions for

cache allocations [4], such as Intel Cache Allocation Technology (CAT) and AMD cache allocation enforcement (CAE) have been widely used in commodity processors to reduce cache interference in clouds. CAT implements way partitioning and provide software interfaces to control cache allocations. Essentially, CAT exacerbates cache conflicts in virtualized systems as the associativity of the cache space is reduced. Many previous works focus on how to better use CAT in existing software systems to reduce cache interference between workloads in clouds [1], [2].

In order to provide transparent and isolated virtual LLCs for VMs, [15] proposes vCache that targets an alternative solution (similar to Intel cache allocation technology) to partition LLC for VMs. vCache presents GPA-based LLC indexing, based on which it implements VM-based LLC partitioning in way granularity to achieve its goals. With vCache, existing cache optimizations in the guest OS take effects but vCache has three main issues. First, vCache needs to change hardware, such that it could not be used directly in commodity processors in clouds. Second, previous studies [5], [34], [38], [39] show that virtual address indexed cache may have worse performance than real physical address indexed cache due to virtual address space changes (e.g., context switches). Third, vCache only targets LLC, and L1 and L2 caches are still indexed by real physical addresses. In modern processors with non-inclusive cache hierarchy, L2 becomes primary and its capacity also becomes larger.

Nested Virtualization. The Turtles Project [11] presents the design and implementation of nested virtualization based on KVM. It mainly addresses the following challenges to realized nested virtualization. First, Intel/AMD only provides one layer of hardware support for virtualization so it is challenging to virtualize CPU in nested VMs. Second, Intel/AMD processors only support two dimensional paging, and it is challenging to support more dimensional page translation for nested VMs; Third, IOMMUs only support a single level of address translation, so it is challenging to translate virtual addresses for devices to physical addresses for nested VMs. To address these challenges, the paper proposes shadow VMCS (virtual machine control structure, VMCB in AMD), EPT on top EPT (Intel extended page table), as well as compressing two levels of translation tables onto the one level that is available in hardware. Some other optimizations such as replacing VMread/VMwrite privileged instructions with binary translation and memory changing with load and store instructions instead of privileged instructions are used to mitigate the number of VMExits in nested VMs.

DVH (direct virtual hardware) [12] mitigates the VMExit multiplication problem in the nested virtulization environment. The basic idea is to let the host hypervisor to emulate virtualized hardware for the nested VM and handle VMExits incurred by the nested VM directly by the emulated hardware instead of forwarding them to the guest hypervisor. The paper introduces four DVH mechanisms: 1) virtual passthrough directly assigns virtual I/O devices to the nested virtual machines; 2) virtual timers transparently remaps timers used by the nested virtual machine to emulated virtual timers provided by the host hypervisor; 3) virtual inter-processor interrupts can be sent and received directly from one nested virtual machines to another; 4) virtual idle enables nested VMs to switch to and from low-power mode without guest hypervisor interventions.

Existing works [11], [12] on nested virtualization mainly focus on realizing nested virtualization and improving its efficiency through reducing the number of VMExits. They do not consider the efficiency of page placement mechanisms in the nested virtualiation environment.

8 Conclusion and Future Work

Nested virtualization becomes increasingly important in today's clouds for the benefits in security, flexibility, and portability that it may bring to systems and applications. However, cache conflicts in the last level cache cause poor cache performance in nested virtualization. This may hamper the wide adoption of nested virtualization in modern clouds, especially cloud workloads becoming more and more memory intensive and the emerging hardware extensions for cache allocation (e.g., Intel CAT and AMD CAE) are used for LLC partitioning.

This work identifies and analyzes this problem in the nested virtualization environment, and proposes XPLACE as an effective system solution. This problem is caused by independent page allocations in different system layer, and must be solved by enhancing the synergy between these layers. Under the nested virtualization setting, the main challenge for achieving synergy is the interposition of guest hypervisors and the portability requirement that limits the solution within the host. XPLACE addresses these challenges by leveraging the property of the page placement mechanism in guest OSs and the shadow page table mechanism.

XPLACE is an effective, efficient, and portable solution. Our evaluations confirm that XPLACE can effectively reduce LLC conflicts to improve application performance and its overhead is low. It does not require the changes to guest OSs or guest hypervisors. Meanwhile, it allows guest OSs to use different page placement mechanisms that can best fit their workloads for higher efficiency.

As future work, we plan to extend and test XPLACE for the system architectures using high bandwidth memory as direct mapped L4 cache. Currently page placement mechanisms and XPLACE are designed for L3 caches. The L3 cache way capacities determine that all huge pages are in the same one or two colors, and page placement in the granularity of huge pages can hardly improve performance. Thus, existing page placement mechanisms, as well as XPLACE, consider only base pages. On the new architectures, with a L4 cache capacity much larger than huge pages, page placement mechanisms must be enhanced to support both base pages and huge pages. XPLACE must be extended accordingly.

References

[1] 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 *EuroSys '18*, 2018, pp. 14:1–14:13.

[2] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: dynamic cache allocation with partial sharing," in *EuroSys '18*, 2018, p. 13:1–13:15.

[3] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee, "vcat: Dynamic cache management using cat virtualization," in *RTAS '17*, 2017, pp. 211–222.

[4] R. Iyer, "Cqos: a framework for enabling qos in shared caches of cmp platforms," in *ICS '04*, 2004, p. 257–266.

[5] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM TOCS*, vol. 10, no. 4, pp. 338–359, 1992.

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

[7] M. Hocko and T. Kalibera, "Reducing performance non-determinism via cache-aware page allocation strategies," in *WOSP/SIPEW '10*, 2010, p. 223–234.

[8] X. Shang, W. Jia, J. Shan, and X. Ding, "Coplace: Effectively mitigating cache conflicts in modern clouds," in *PACT '21*, 2021, p. 274–288.

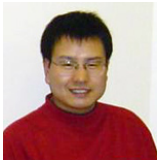
[9] X. Ding, K. Wang, and X. Zhang, "Ulcc: a user-level facility for optimizing shared cache performance on multicores," in *PPoPP '11*, 2011, p. 103–112.

[10] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Enabling software management for multicore caches with a lightweight hardware support," in *SC '09*, 2009, p. 14:1–14:12.
[11] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI '10*, 2010, p. 423–436.
[12] J. T. Lim and J. Nieh, "Optimizing nested virtualization performance using direct virtual hardware," in *ASPLOS '20*, 2020, p. 557–574.
[13] D. Williams, H. Jamjoom, and H. Weatherspoon, "The xen-blanket: Virtualize once, run everywhere," in *EuroSys '12*, 2012, p. 113–126.
[14] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *SOSP '11*, 2011, p. 203–216.
[15] D. Kim, H. Kim, N. S. Kim, and J. Huh, "vCache: Architectural support for transparent and isolated virtual llcs in virtualized environments," in *MICRO '15*, 2015, p. 623–634.
[16] "The Princeton application repository for shared-memory computers (PARSEC)," http://parsec.cs.princeton.edu/, 2010.
[17] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *IISWC '16*, 2016, p. 1–10.
[18] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, no. 10, pp. 623–624, 1965.
[19] D. Knuth, "The art of computer programming: Fundamental algorithms, volume 1. addisonwesley," *Reading, Mass.,*, 1997.
[20] D. G. Korn and K.-P. Bo, "In search of a better malloc," in *Proceedings of the Summer 1985 USENIX Conference*. USENIX, 1985, pp. 489–506.
[21] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.
[22] "Kernel Samepage Merging," https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.
[23] C. A. Waldspurger, "Memory resource management in vmware esx serve," in *OSDI '02*, 2002, p. 181–194.
[24] J. Park, H. Yeom, and Y. Son, "Page reusability-based cache partitioning for multi-core systems," *IEEE Trans Comput*, vol. 69, no. 6, pp. 812–818, 2020.
[25] "Intel VMCS Shadowing," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf.
[26] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.
[27] "A deep network handwriting classifier," https://github.com/xingdi-eric-yuan/multi-layer-convnet.
[28] 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.
[29] 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 *ACL '07 on interactive poster and demonstration sessions*, 2007, p. 177–180.
[30] "Xapian project," https://github.com/xapian/xapian.
[31] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys '12*, 2012, p. 183–196.
[32] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP '13*, 2013, p. 18–32.
[33] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-mt: a scalable storage manager for the multicore era," in *EDBT/ICDT '09*, 2009, p. 24–35.
[34] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice — a low-cost high-speed address translation mechanism," in *ISCA '90*, 1990, p. 355–363.
[35] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches," *IEEE Trans Comput*, vol. 38, no. 12, pp. 1612–1630, 1989.
[36] 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 *HPCA '08*, 2008, p. 367–378.
[37] H. Kim and R. R. Rajkumar, "Predictable shared cache management for multi-core real-time virtualization," *ACM Trans. Embed. Comput. Syst.*, vol. 170, no. 1, pp. 22:1–22:27, 2017.
[38] W.-H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *ISCA '89*, 1989, p. 140–148.
[39] A. J. Smith, "Cache memories," *ACM Comput. Surv*, vol. 14, no. 3, pp. 473–530, 1982.

Xiaowei Shang is a PhD student in Computer Science at New Jersey Institute of Technology. Her research interests include computer systems and architecture with a focus on CPU cache optimizations for virtualization environments.

Weiwei Jia is an Assistant Professor in the Department of Electrical, Computer, and Biomedical Engineering at The University of Rhode Island. He received his Ph.D. degree in Computer Science from New Jersey Institute of Technology. His research interests lie in the areas of computer systems, including operating systems, edge and cloud computing, virtualization, memory and storage systems, and systems architecture.

Jianchen Shan received a Ph.D. degree in Computer Science from New Jersey Institute of Technology. He is an Assistant Professor at Hofstra University. His research interests include Cloud Infrastructure, High-Performance Computing, and Operating System.



Xiaoning Ding is an Associate Professor at New Jersey Institute of Technology. His interests are in the area of experimental computer systems, such as distributed systems, virtualization, operating systems, and storage systems. He earned his Ph.D. degree in computer science and engineering from the Ohio State University.

Cristian Borcea is a Professor in the Department of Computer Science at NJIT. He is also a Visiting Professor at the National Institute of Informatics in Tokyo, Japan. His research interests include mobile computing and sensing, ad hoc and vehicular networks, distributed systems, and cloud computing. Borcea received his Ph.D. degree from Rutgers University in 2004. He is a member of the ACM and IEEE.