vCache: Architectural Support for Transparent and Isolated Virtual LLCs in Virtualized Environments

Daehoon Kim*, Hwanju Kim†, Nam Sung Kim*, and Jaehyuk Huh‡
*University of Illinois, Urbana-Champaign, †University of Cambridge, ‡KAIST
{daehoonk, nskim}@illinois.edu, Hwanju.Kim@cl.cam.ac.uk, jhhuh@kaist.ac.kr

ABSTRACT

A key role of virtualization is to give an illusion that a consolidated workload runs on a dedicated machine although the underlying resources are actively shared by multiple workloads. Technical advances have enabled a virtual machine (VM) to exercise many shared resources of a machine in a transparent and isolated manner. However, such an illusion of resource dedication has not been supported for the last-level cache (LLC), although the LLC is the largest on-chip shared resource with a significant performance impact. In this paper, we propose vCache–architectural support to provide a transparent and isolated virtual LLC (vLLC) for each VM and interfaces to manage the vLLC. More specifically, this study first proposes architectural support for the guest OS of a VM to index the LLC with its guest physical address instead of a host physical address. This in turn allows that the guest OS transparently view its vLLC and preserve the effectiveness of its page placement policy. Second, this study extends the architectural support for each VM to keep its vLLC strongly isolated from other VMs. Such resource dedication is critical to offer performance isolation and preserve vLLC transparency for each VM in a highly consolidated machine. With little hardware overhead, vCache can facilitate various unchartered vLLC capacity-based services for the public clouds while providing up to 17% higher performance than a traditional virtualized system.

Categories and Subject Descriptors

B.3.2 [**Memory Structure**]: Design Styles—*Cache memories*; C.1.0 [**Processor Architectures**]: General

Keywords

virtualization, virtual LLC, performance isolation, page coloring

MICRO-48, December 05-09, 2015, Waikiki, HI, USA
 ISBN 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830825

1. INTRODUCTION

Diverse computing environments ranging from data centers to embedded systems have adopted machine virtualization to efficiently consolidate multiple workloads on a single physical machine. A key role of machine virtualization is to provide an illusion that each workload runs on a dedicated machine, while sharing machine resources with other co-executed workloads [1]. In particular, such an illusion should be rigorously provided for utility computing that is typically offered by the public clouds where each user desires his/her workload to fairly run on the paid resources. In this regard, there have been many studies to efficiently isolate resources among consolidated VMs [2, 3, 4, 5, 6, 7, 8]. Creating an illusion of dedicated resources for each VM must consider two aspects – *transparency* and *isolation*. The transparency permits each VM to fully control its resources, as if the guest OS of each VM manages its dedicated resources, while the isolation shuns any possible performance interference amongst VMs.

Although virtualization techniques have offered the illusion of dedicated resources for many sharable resources such as CPU, memory, and I/O devices, one of the neglected on-chip resources yet to be virtualized is the LLC due to the lack of architectural support. The LLC, shared by multiple cores and thus VMs, is an expensive resource due to its considerable and increasing footprint on a die; LLCs consume nearly 50% of die area in most commercial processors (e.g., Intel Sandy Bridge Xeon E3 processors with 8MB [9]) and their capacity has significantly increased (e.g., Intel Haswell Xeon E5 processors with 45MB LLC [10]).

Need for transparent and isolated vLLC: The OS often desires to manage the placement of memory pages in the LLC by manipulating the LLC index portion of a physical address for higher utilization, lower pollution, and more fairness. A classic example is page coloring [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Such an OS-level technique, however, becomes ineffective in current virtualized systems, since a guest physical address (GPA) managed by a guest OS diverges from a host physical address (HPA) used to index the LLC due to the hypervisor's GPA-to-HPA mapping [22, 23]. Besides, due to the lack of architectural support for isolating vLLCs, the hypervisor cannot explicitly control the vLLC capacity for individual VMs. Consequently,

the performance of a VM may suffer from interference and contention caused by co-running highly memory-intensive VMs [4, 24].

Key contributions: In this paper, we present *vCache*– architectural support for transparent and isolated vLLCs and interfaces for the vLLC management. Our key contributions are as follows:

- First, we propose architectural support for a transparent vLLC after demonstrating that page coloring becomes ineffective in a traditional virtualized system. This architectural support allows a guest OS to index the LLC with its GPA directly so that the guest OS can control the placement of memory pages in the LLC, as if the OS manages the LLC in a non-virtualized system.

- Second, we extend the architectural support to provide strong isolation amongst vLLCs and guarantee the minimum vLLC capacity for a VM. The minimum vLLC per VM can mandated by the contract with its user. Moreover, we demonstrate that providing such a strong isolation amongst vLLCs is critical to preserve the transparency of vLLCs in a multi-VM system.

- Lastly, we explore the potential of vLLC capacity allocation policies using the vLLC management interface of *vCache*. Then we demonstrate that the transparency provided by *vCache* plays a critical role to further improve the performance over a traditional LLC partitioning technique applied to a multi-VM system.

Note that *vCache* can open up various business and research opportunities by (1) facilitating vLLC capacity-based services for the public clouds such as Amazon EC2 and (2) allowing each guest OS to deploy its own LLC management policy in a multi-VM system. Regarding (2) the guest OS can determine a better LLC management policy since the guest OS has more diverse interfaces with applications and understand its applications better than the hypervisor or hardware. Lastly, *vCache* does not require any guest OS changes with its transparency. It requires only small changes in on-chip cache architecture and a slightly modified hypervisor to support page sharing and enforce LLC allocation policies.

Comparison with alternative approaches: First, the hypervisor can preserve colors by allocating large contiguous regions for VMs. However, this restricts the memory allocation flexibility of the hypervisor. For example, using a large page size for the guest physical memory allocation can partially preserve colors, but it restricts the fine-grained memory management by the hypervisor, and the interference among consolidated VMs still exist or become worse, if all VMs compete for certain limited colors in physical LLCs without any control. In contrast, *vCache* neither imposes such a restriction on allocating pages nor incurs the overhead of re-coloring the pages.

Second, *vCache*, which partitions the LLC per VM, partly shares the underlying mechanism with prior core-based LLC partitioning techniques such as utility-based cache partitioning (UCP) [25]. Nonetheless, due to the support of both transparency and isolation together, *vCache* can offers superior performance and fairness to existing LLC partitioning techniques.

Lastly, although NUMA-aware hypervisors attempt to mitigate the contention problem by controlling thread scheduling on multiple sockets [26], such a NUMA scheduling policy addresses only inter-LLC scheduling with multiple LLCs, and it attempts to maximize only the throughput in a best-effort manner without supporting transparency and isolation.

Advantage over prior HW partitioning: A key advantage of *vCache* is that it can support independent LLC managements by guest operating systems and the hypervisor. Prior partitioning schemes and commercial realization allow partitioning which can be assigned to either application thread or VM, but not both. However, *vCache* allows the guest OS to manage its vLLC with page coloring, and the hypervisor can control the allocation of vLLCs to VMs. Such a dual-layer approach enables the guest OS controlled by a cloud user to manage its own vLLC with better information on its applications than the hypervisor. Simultaneously, the cloud administrator can set a per-VM allocation policy to provide consistent LLC performance for each user VM.

Effectiveness of *vCache*: To evaluate the efficacy of *vCache*, we use the SIMICS full-system simulator and Xen to run consolidated workloads, each of which runs on a VM. First, we show that *vCache* can preserve the effectiveness of page coloring policies directed by the guest OS through its transparency support. For example, *vCache* improves performance of *hmmer*, which co-runs with *libquantum*, by 17% compared to a traditional virtualized system. Second, we demonstrate that *vCache* can deliver almost the same performance as or even better performance than VMs with separate physical LLCs per VM through its vLLC isolation capability with work-conserving policy. Lastly, we show that *vCache* further improve performance with vLLC capacity allocation policies. For example, *vCache* increases overall performance improvement of a capacity allocation policy for three co-running applications (*hmmer*, *libquantum*, and *matmul*), from 2% to 12%, compared to a traditional HPA-indexed LLC without the policy.

Organization: The remainder of this paper is organized as follows. Section 2 describes motivation and backgrounds. Section 3 proposes *vCache* architecture and analyzes its hardware overhead. Section 4 discusses additional implementation issues to support *vCache*. Section 5 evaluates *vCache*. Section 6 describes related work, and Section 7 concludes the paper.

2. MOTIVATION AND BACKGROUNDS

2.1 Transparency for LLC

Since most commercial processors use some of physical address bits to index a set in their LLC, the OS can

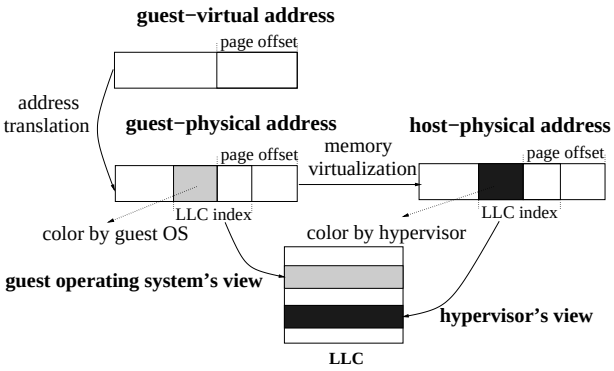


Figure 1: The impact of memory virtualization on page coloring by a guest OS.

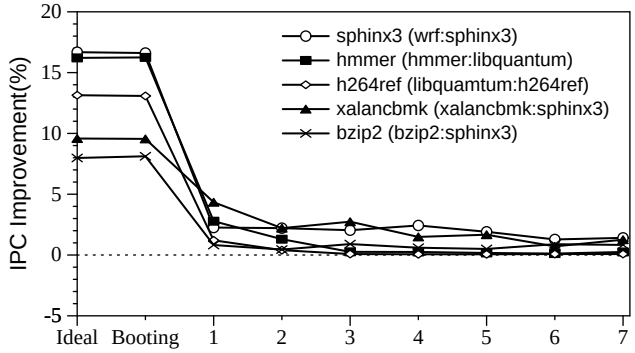


Figure 2: The effectiveness of the pollute buffer mechanism in a virtualized system.

apply a technique to control where a memory page is placed in the LLC by manipulating its physical address, also known as page coloring. With page coloring, the OS partitions the physical memory space into multiple colors, choosing a color to place memory pages in certain contiguous LLC sets during the allocation of a memory page. Such a page coloring technique can allow the OS to evenly spread accesses across LLC sets or reduce interference among multiple applications sharing the LLC by assigning a suitable LLC region for each application. Thus, it is widely supported by the mainstream operating systems such as Solaris, FreeBSD, netBSD, and Windows NT [12, 19, 13, 20, 17].

In virtualized systems, however, a guest OS can no longer precisely control the placement of its memory pages using such an OS-driven page coloring technique because of another indirection incurred by memory virtualization. That is, the hypervisor maps a GPA assigned by the guest OS to an HPA (i.e., a physical address used to index an LLC set) when a VM is created. Figure 1 illustrates how the virtualization alters a GPA's page color assigned by a guest OS. Although the guest OS expects the virtual memory page to be mapped to the gray region of the LLC, the hypervisor actually places the memory page at the black region of the LLC due to the GPA-to-HPA mapping, losing the original page color assigned by the guest OS. To the best of our knowledge, the mainstream hypervisors such as Xen and VMware workstation do not preserve page colors assigned by guest operating systems, making page coloring techniques ineffective.

Basic management commands to consolidate and manage VMs change the GPA-to-HPA mapping, such as boot/destroy and suspend/resume. For example, after a hypervisor first creates a VM, the page colors can be mostly preserved since the hypervisor often attempts to allocate large contiguous real physical memory regions for the VM. As many VMs are frequently created, stopped, and resumed in highly consolidated systems, the physical memory managed by the hypervisor is fragmented, and the hypervisor cannot always provide large contiguous memory regions for VMs. This makes the hypervisor unable to preserve colors assigned by guest operating systems.

Furthermore, the hypervisor dynamically assigns and adjusts the memory resources of running VMs. This leads to further memory fragmentation and changes in GPA-to-HPA mapping. The schemes for dynamic memory management such as memory ballooning [23] and live migration [27] have been commonly used for highly consolidated systems, since the memory is a primary factor limiting a VM consolidation ratio. For example, as one VM demands more memory space, the hypervisor dynamically allocates more pages to the VM by stealing some pages from another VM with sufficient memory at runtime (i.e., memory ballooning). This alters the existing GPA-to-HPA mapping. Finally, the hypervisor migrates a VM from one (overloaded) physical machine to another (underloaded) physical machine (i.e., VM migration). This creates a new GPA-to-HPA mapping.

As an example, we take the pollute buffer mechanism [17], a cache management technique leveraging page coloring to alleviate an LLC pollution problem, to demonstrate how page coloring by a guest OS becomes ineffective in a traditional virtualized system. The pollute buffer mechanism observes the re-usability of pages for an application, and then dynamically remaps cache-unfriendly pages to a small pollute region in the LLC. This minimizes competitions between pages with high and low re-usability. Later, we will employ another page coloring technique (ULCC) for more extensive evaluation of *vCache*.

We use an evaluation environment based on Xen [28] running on the SIMICS [29] full-system simulator. To support 128 different page colors, we assume that the page size is 4KB (i.e., 12 bits for the page offset) and the guest OS can manipulate 7 most significant bits of 13 bits used to index a 4MB 8-way LLC with 64B block size. We implement the pollute buffer mechanism by emulating the OS policy in the simulator and allocate four colors for pollute pages. If the miss rate of a page is higher than a threshold, the page is classified as a pollute page, and mapped to the pollute color. To compare the effectiveness of this pollute buffer mechanism in a virtualized system, we set up a VM with two virtual CPUs (vCPUs) and enable the memory ballooning

technique. The VM runs a mix of two applications from the SPEC CPU2006 suite.

Figure 2 plots the performance of a virtualized system in terms of normalized IPC as memory ballooning occurs; y-axis shows the performance of a virtualized system using the pollute buffer mechanism, relative to that of a system without the pollute buffer mechanism and x-axis shows the number of ballooning occurrences right after a VM is booted. The experiments assume that the physical machine initially has a large contiguous memory region to allocate for the VM. Figure 2 demonstrates that the effectiveness of the pollute buffer mechanism considerably degrades with more memory ballooning occurrences. When the VM is booted, the GPA-to-HPA mapping is close to the ideal case since the hypervisor initially has a large contiguous free memory region. However, the effectiveness of the pollute buffer mechanism rapidly decreases with more memory ballooning occurrences. This is because the hypervisor ends up remapping pages belonging to a certain (GPA) color to various (HPA) colors. For example, the pollute buffer mechanism can improve the performance of *sphinx3* co-running with *wrf* by 17% when the original page colors can be preserved. However, even after the first memory ballooning occurrence, the performance improvement by the pollute buffer mechanism is decreased to 2% because the original page colors are lost due to the aforementioned remapping process. After one or two more memory ballooning occurrences, the pollute buffer mechanism becomes practically ineffective. In summary, our experiment shows that memory mapping changes due to virtualization can significantly disturb the original placements of memory pages in the LLC arranged by a guest OS and make the positive effect of page coloring mostly disappear.

2.2 Isolation for LLC

Since the LLC plays an important role of absorbing expensive off-chip memory traffics, it has a significant impact on performance and energy consumption, especially for applications sensitive to the LLC size. As the LLC is typically shared by multiple cores (and thus multiple VMs), performance interference by co-running applications is inevitable. In a virtualized system, usually multiple tenants are hosted. Hence, it is desirable to provide a mechanism to isolate the use of a shared LLC and prevent a VM from suffering from unexpected and non-deterministic performance degradation by the interference.

One simple method to provide a VM with an isolated LLC is to dedicate fixed physical cores and LLC slices to a single VM; a physical core is associated with a slice of LLC (e.g., a core has a 2MB LLC slice in an octa-core processor with 16MB LLC [30]). However, this approach is neither effective nor flexible. The demand on cores is not always correlated to the demand on LLC capacity. A single-threaded application may require a large LLC capacity, while a multi-threaded application has a small cache working set. Therefore, to virtualize the LLC effectively, *vCache* must support a mechanism to adjust the LLC capacity for each VM, while supporting transparency.

Without such isolation support, transparency alone cannot provide the benefit of the OS-level page coloring in consolidated systems. For example, with the pollute buffer mechanism, each guest OS may assign a different color for its pollute buffer, and when multiple VMs share a physical LLC, the pollute buffer data of a VM can evict important cache lines of normal regions of other VMs. To truly support a vLLC for each VM with the property of a dedicated resource, transparency and isolation must always be provided together.

Prior LLC isolation supports: There have been several efforts to preserve LLC-related quality of service (QoS) by mitigating inter-VM interferences [24, 4]. *Q-Clouds* pointed out that desired performance may not be guaranteed due to LLC interference [4]. To ensure QoS, it adjusts CPU allocation based on estimated performance interferences instead of directly controlling LLC allocation. *Cuanta* proposed a technique for predicting performance interference due to the shared LLC [24].

For virtualized systems, the hypervisor may directly enforce page coloring for VMs or applications to control cache allocation, but enabling *vCache* does not allow such hypervisor-based page coloring. However, the rationale for *vCache* approach is to allow independent cache managements by the guest OS and hypervisor. With *vCache*, the guest OS with better information about its application, controls cache placements for the application with page coloring. The hypervisor controls per-VM cache allocation with the support for LLC isolation. This dual-layer approach is one of the advantages of *vCache* over prior partitioning techniques.

3. VCACHE ARCHITECTURE

This section presents the proposed *vCache* architecture which provides a transparent and isolated vLLC for each VM. The section also analyzes its hardware overhead in terms of area, power, and timing.

3.1 Transparency: GPA-based LLC Indexing

vCache can provide a transparent view of LLC by allowing each guest OS to preserve its original page coloring as if it is running on a non-virtualized system. Such transparency is supported by maintaining page color bits of GPAs and incorporating them to index an LLC. Figure 3 illustrates the *vCache* architecture and associated minor changes in the TLB and cache architectures. The figure assumes an 8-way associative LLC of 4MB size with 64B blocks (i.e., 13 and 6 bits for set index and block offset) and 1TB physical memory size with 4KB pages (i.e., 40 and 12 bits for physical memory address and page offset).

A key mechanism of *vCache* to support transparency is to index the LLC with GPA. However, only a host physical page address (HPA[39:12]) is stored in each TLB entry of conventional architecture. For the GPA-based LLC indexing, each TLB entry in *vCache* is extended to store page color bits of a GPA (GPA [18:12])

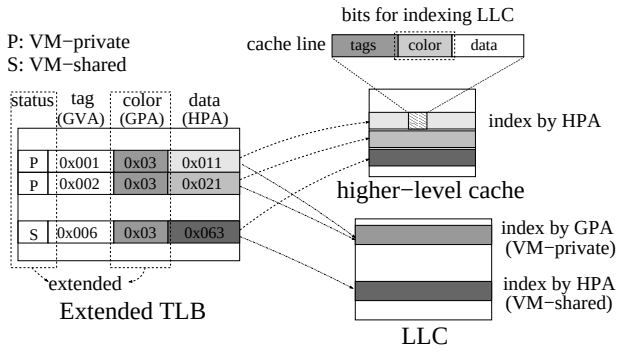


Figure 3: *vCache* architecture.

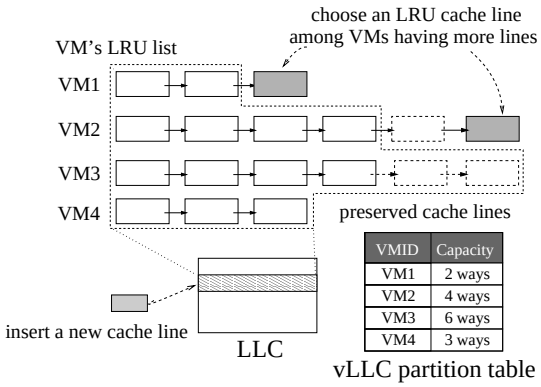


Figure 4: Replacement policy of *vCache*.

as well. In general, a TLB needs $\log_2 n$ more bits for each entry where n is the number of page colors. For example, *vCache* augments 7 bits per TLB entry for 128 page colors. Upon a TLB miss, both a physical page address (HPA[39:12]) and the page color bits of the corresponding GPA are stored in each TLB entry after a page table walk. To generate the LLC set index, the page color bits, which were obtained during a TLB look-up, are concatenated with part of the given GVA ({GPA [18:12], GVA[11:6]} = GPA[18:6]), as illustrated in Figure 3. Since GVA, GPA, and HPA have the same page offset (GVA[11:0] = GPA[11:0] = HPA[11:0]), this effectively allows *vCache* to index the LLC based on a GPA, and thus to preserve the original page coloring of the guest OS.

Although the LLC is indexed by GPA, the actual tags are matched with HPA to avoid aliasing across VMs and extra translation during write-backs. For tag matching based on HPA, each LLC tag is extended to store the entire physical page address (HPA[39:12]) in *vCache*, unlike conventional architectures which store only part of physical page address (HPA[39:19]) in the tags. Such an LLC tag extension is needed for two reasons. First, by using the HPA tags, cache lines with the same GPA from different VMs can be distinguished. Second, if a dirty LLC block has to be evicted and written back to the main memory in conventional architecture, HPA[39:6] is reconstructed from its tag value and set index producing HPA[39:19] and HPA[18:6], respectively. However, in *vCache*, HPA[18:12] cannot be inferred by the set index of the evicted one, since the set index is based on GPA[18:12] (GPA[18:12] $\neq$ HPA[18:12]), and thus must be stored in the tags.

Higher-level caches such as L1 and L2 caches do not require the aforementioned steps to generate a set index as done for the LLC, because they are indexed by HPA. However, when an L2 cache block is evicted and written back to the LLC, the color bits are required to index the LLC. Hence, each tag of higher-level caches is also extended to store the color bits of the corresponding GPA.

In virtualized systems, however, using a GPA to index the LLC is not always possible. First, the memory space used by the hypervisor itself may not have corresponding GPA since it is not part of a virtualized memory space. Furthermore, two different pages in GPA spaces can be mapped to a single page in the HPA space for page sharing. For those shared pages, GPA cannot be used for indexing the LLC, since a potential aliasing problem can occur.

To avoid such a problem, all the pages are distinguished to either *VM-private* or *VM-shared* pages by a page status bit. A VM-private page is a guest physical page with one-to-one mapping between the guest physical page and the host physical page, and it is indexed by GPA as described earlier in this section. In contrast, a VM-shared page is a page used by either the hypervisor or a guest physical page shared by multiple VMs or within a VM. It is indexed by only HPA in the LLC. Since the hypervisor always has to know whether a page is VM-private or VM-shared, it maintains the page status in the page table entry. However, if a VM-private page stored in the LLC becomes a shared page, or vice versa, duplicated copies can be loaded, and thus, the existing LLC blocks must be flushed when the page status bit is updated.

3.2 Isolation: VM-based LLC Partitioning

In order to provide isolated vLLC, *vCache* implements a VM-based cache partitioning in way granularity, which allows each VM to leverage distinct cache ways on a set-associative cache. The way-based partitioning can divide a large physical set-associative cache into multiple smaller caches in way granularity; For example, using the way-based partitioning, an 8MB, 16-way set-associative cache can be partitioned into four small caches: 1MB 2-way, 2MB 4-way, 4MB 8-way, 1MB 2-way set-associative caches. If there are four VMs and we assign each partitioned cache to each VM, a VM can use its isolated cache without interference by other VMs. Note that the associativity and capacity have increased with more cores per processor (e.g., a 10-core Intel Xeon E7-8800/4800/2800 processor with a 30-way 30MB LLC).

To support the VM-based partitioning in way granularity, *vCache* uses a hardware table, called *vLLC partition table*, to specify the number of ways for each VM. The vLLC partition table simply records VM identifiers and vLLC capacities in terms of ways. *vCache* looks up

the vLLC partition table to check and keep the capacity of each vLLC when evicting and loading a cache line. The maximum capacity that a VM can use and the sum of all vLLC capacities must be equal to or smaller than the entire physical LLC capacity.

For the VM-based partitioning, *vCache* employs a modified LRU replacement policy similar to prior core-based LLC partitioning techniques [25, 31, 32]. When inserting a cache line from the LLC, *vCache* controls the number of cache lines at each cache set by the way limit in the vLLC partition table. For such control, each cache line keeps a VM identifier to identify the VM that loaded the cache line. For an LLC miss, *vCache* chooses an LRU cache line as a victim amongst the VMs with more cache lines than their partition. Otherwise, *vCache* chooses an LRU cache line amongst all loaded cache lines as a conventional LRU-based replacement algorithm. Figure 4 briefly describes how *vCache* chooses a victim cache line, when inserting a new cache line. The four vLLCs from VMs share the physical LLC, and the capacity of each vLLC is set by the vLLC partition table. A victim cache line is selected from the extra cache lines occupied by VM1 or VM2 by the LRU policy.

For high LLC utilization, *vCache* employs a work-conserving policy similar to the one proposed by Nesbit et al [33]. If the sum of partitions reserved by all VMs is smaller than the physical LLC, the unreserved capacity is shared by the VMs with demand-based allocation. Furthermore, as a further optimization for improving LLC efficiency, if a VM becomes idle for a long period of time, the reserved ways for the VM can be reduced or eliminated. Redistributing the capacity for an idle VM is simple. By resetting the partition table value for the VM, the released cache capacity will be occupied by other VMs, as they incur cache misses. Although *vCache* provides a framework that shares idle partition, detecting temporal idleness of vLLCs remain as our future work.

A VM-based cache management requires a supplementary mechanism for VM-shared pages that do not belong to any VM. To handle such VM-shared pages in a simple way, *vCache* treats a newly loaded cache line of a VM-shared page as a cache line of the evicted VM and chooses an LRU cache line as a victim. With this policy, VM-shared pages can be loaded into any position regardless of vLLCs, and the shared cache lines are controlled by the same replacement algorithm as VM-private pages.

vCache focuses on providing a transparent vLLC that can preserve the expected LLC performance with isolation to each VM. Since *vCache* is proposed to guarantee the capacity of each vLLC in this paper, we do not investigate *dynamic* vLLC capacity allocation policies that may disturb preserving the transparency of vLLCs. However, if improving the overall combined throughput is critical, *vCache* can employ prior LLC allocation policies [25, 31] at VM granularity. Although we do not propose a new policy to determine the best way partition for a given VM consolidation scenario, we explore the potential of *vCache* employing static vLLC capacity allocation policies for high overall performance in Section 5.3.

3.3 Hardware Overhead

Since *vCache* requires some changes in the TLB and cache architecture, we evaluate its impact on power and timing in this section. Each entry of TLBs needs $\log_2 n$ more bits where n is the number of page colors and an additional 1 bit for page status. Each tag of the LLC and its higher level caches must have extra $\log_2 n$ bits for the extra part of HPA bits. Note that the extra $\log_2 n$ bits in each LLC tag participates in tag comparison, while ones in each higher level cache tag do not. Each LLC line requires $\log_2 m$ bits to keep a VM identifier where m is the number of VMs. In addition, each LLC set requires $\log_2 k$ bits per VM for a counter to indicate how many ways are allocated for each VM where k is the number of ways supported by *vCache*. Finally, *vCache* needs a vLLC partition table with and $\log_2 k$-bit m entries to maintain vLLC to VM mapping.

For our analysis using McPAT [34] and CACTI [35], we assume a processor configuration similar to Sandy Bridge with 128 page colors ($n = 128$), 4 VMs ($m = 4$), and 16-way 8MB LLC ($k = 16$). For L1 and L2 caches, we use high-performance and low-standby devices for peripheral circuits and memory cells, respectively. In addition, LLC, we use low-standby devices for peripheral circuits and memory cells, assuming a sequential mode for tag and data array accesses. Our analysis shows that the adding these extra bits for TLBs, L1 and L2 caches do not affect the processor cycle time. In contrast, the extra bits for the LLC participate tag comparisons, increasing the number of access cycles by one at most. *vCache* increases the dynamic energy per access (static power) of L1 caches, L2 cache, LLC, and TLBs, by 0.3% (0.9%), 0.5% (0.7%), 0.8% (2.1%), and 11% (6%) respectively. Finally, *vCache* increases the area of TLBs, L1 caches, L2 cache, LLC, TLBs by 0.7%, 0.7%, 1.8%, and 16%, respectively. Considering the power consumption and area of these blocks relative to that of the total multi-core processor, we can see that the overall power increase is negligible.

4. IMPLEMENTATION ISSUES

4.1 Hypervisor Support for vCache

For VM-based LLC management, *vCache* utilizes the VM information in the hypervisor layer, such as VM identifier, but it does not require any significant modification of hypervisors for its basic functionality. Current hardware-assisted virtualization techniques already maintain in-memory data structures (e.g., virtual machine control structure (VMCS)) per VM to save/restore VM states for transitions between root and non-root mode. To distinguish a scheduled VM, *vCache* uses the page table pointer to the VM address space (nCR3 in x86). For LLC isolation, the hypervisor must set the required vLLC capacity for partitioning in the VMCS, and update the partition table, if any adjust-

ment for LLC partitioning is necessary. To distinguish VM-private and VM-shared, hypervisor must keep the sharing status of memory pages in each page table entry. However, current hypervisors already track the sharing status of each page for memory management. Thus, for *vCache*, its hypervisor just stores the already tracked status bit (1 bit) in the page table entry. Although a slight modification of the hypervisor is required to support *vCache*, guest operating systems do not need to be modified.

4.2 Cache Coherence and DMA

vCache requires minor modifications on coherence protocols to support cache coherence across *multiple* LLCs for VM-private pages. Since *vCache* uses GPA for indexing LLCs, traditional coherence protocols with HPA cannot look up a correct set in the target LLC. To support such inter-LLC cache coherence, each coherence transaction must carry the extra color bits from GPA in addition to the full HPA. The GPA color bits are used to find the correct set in the target LLC, while the full HPA is used to determine the corresponding cache line in the set through tag matching. All steps for the coherence transaction are the exactly same with the traditional coherence protocols except for finding the proper set using GPA. For VM-shared pages, coherence operations do not require such GPA-based indexing.

For DMA devices, the hypervisor must set pages for DMA as VM-shared, so that coherence requests from a DMA device can look up the correct set in the cache with HPA. Since the hypervisor identifies which pages are cacheable and used for DMA, it can properly set those pages as VM-shared. Alternatively, if the DMA device uses IOMMU, *vCache* can store DMA pages with GPA-based indexing in the LLC. In that case, since I/O page tables can use GPA as virtual address of the device in virtualized systems, the extension for IOTLB is not required to store GPA colors.

4.3 Hashed/Randomized LLC slices

To increase available bandwidth and reduce contentions, some multi-core architectures (e.g. Intel Sandy Bridge and Ivy Bridge architectures) organize the LLC architecture with multiple slices, and use a hashing function to evenly distribute cache lines across the LLC slices. With the hashed LLC, the color range available to the OS is reduced by a factor of the number of slices. For example, a non-hashed LLC has 128 colors, but a hashed LLC with the same capacity consisting of four slices can only support 32 colors. In that case, the OS can use five bits, instead of 7 bits, for page color. Cache lines from the same color can be spread to any slices, but will be located in the same set of the slices.

However, in an extremely hashed LLC where the set index is completely hashed from different bit portions of address, the OS will have difficulty with employing page coloring unless it understands the hash mapping function. If the hash function uses part of physical address higher than page offset, and the function is known, the guest OS can control page placement across slices by restoring coloring bits. Further investigation on restoring colors for multiple slices will be our future work.

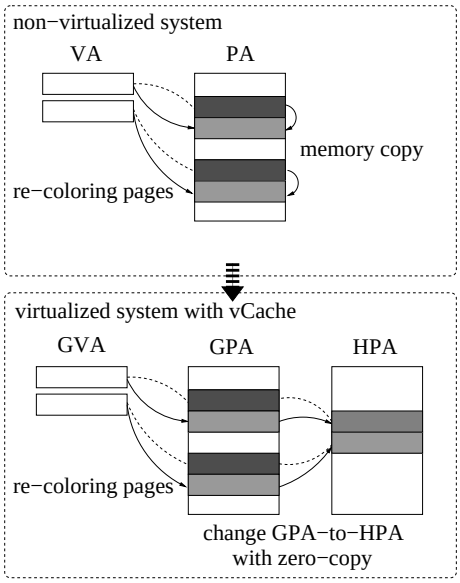


Figure 5: Zero-copy re-coloring with *vCache*.

4.4 Zero-copy Re-coloring

In addition to preserving page colors set by the guest OS, *vCache* can reduce the cost of page re-coloring in the traditional page coloring techniques. In particular, the cost for memory copies during the re-coloring process is a primary reason that prevents a dynamic page coloring technique from achieving its full potential performance improvement [19]. Figure 5 describes the page re-coloring processes of non-virtualized systems and virtualized systems with *vCache*. In non-virtualized systems, to change the color of a virtual page, its physical page must be changed to another physical page with a different color. During the re-coloring process, the page content must be copied to a new page, causing some delay. Therefore, dynamic re-coloring to adapt to phase changes of applications can result in unexpected performance degradation. Although some of the costs can be hidden by processing re-coloring in the background, it still consumes notable CPU cycles and memory bandwidth.

In contrast, the re-coloring process with *vCache* is simple. The color of a page is set with the bits in GPA, and thus changing only the GPA without changing the machine page can effectively change the LLC region for the page. The guest OS picks another page with a demanded page color in its guest physical address space, and requests the hypervisor to map the new guest page to the original machine page. For coherence, dirty blocks for re-colored pages in the LLC should be explicitly flushed with the old GPAs in the pages. Since the machine page never changes during the re-coloring process, the guest OS can change the color of a virtual page without the page copy cost by simply changing

Table 1: SPEC CPU2006 mixes for pollute buffer mechanism.

abbrv.	workload mixes	abbrv.	workload mixes
HM-LI	hmmer, libquantum	WR-SP	wrf, sphinx3
LI-H2	libquantum, h264ref	BZ-SP	bzip2, sphinx3
SO-CA	soplex, calculix	BZ-SO	bzip2, soplex
HM-SO	hmmer, soplex	XA-SP	xalancbmk, sphinx3
AS-SO	astar, soplex	XA-LB	xalancbmk, lbm
SP-GR	sphinx3, gromacs	XA-ZE	xalancbmk, zeusmp

the GPA-to-HPA page mapping and explicit flushes of dirty blocks. Although the re-coloring request requires hypervisor intervention, the overheads could be alleviated by collectively batching multiple mapping updates. With the zero-copy re-coloring, the page coloring policy of each guest OS can efficiently adapt to phase changes of applications.

5. EVALUATION

In this section, we evaluate *vCache* in three key aspects. First, we show that *vCache* provides a transparent vLLC to a VM. Such a transparency allows the guest OS to preserve the effectiveness of its page coloring even after memory virtualization. Second, we demonstrate that *vCache* provides an isolated vLLC for each VM while preserving the transparency of each vLLC in a multi-VM system. This in turn allows each VM to run its own unique page coloring algorithm. Lastly, we show that *vCache* can support diverse inter-VM vLLC capacity allocation policies to embrace various workload demands.

To evaluate *vCache*, we use two page coloring techniques: pollute buffer mechanism and user level cache control (ULCC) [21]. For the pollute buffer mechanism, we take benchmark mixes from SPEC CPU2006 with the *ref* input sets. Table 1 tabulates the abbreviation of each benchmark. We run two benchmarks concurrently in a VM with two vCPUs for 1B instructions per benchmark after warm-up cycles. To evaluate the page coloring effectiveness, we pin each benchmark to a vCPU, and then we pin each vCPU to a separate physical core. ULCC is a software runtime library that allows a programmer to explicitly allocate proper page colors to various data structures. Using the ULCC interface, programmers are capable of coloring pages for given application's characteristics. For ULCC, we run each PLUTO benchmark [36] (i.e., *matmul, dct, covcol, and mvt*) in a VM with a vCPU for 1B instructions.

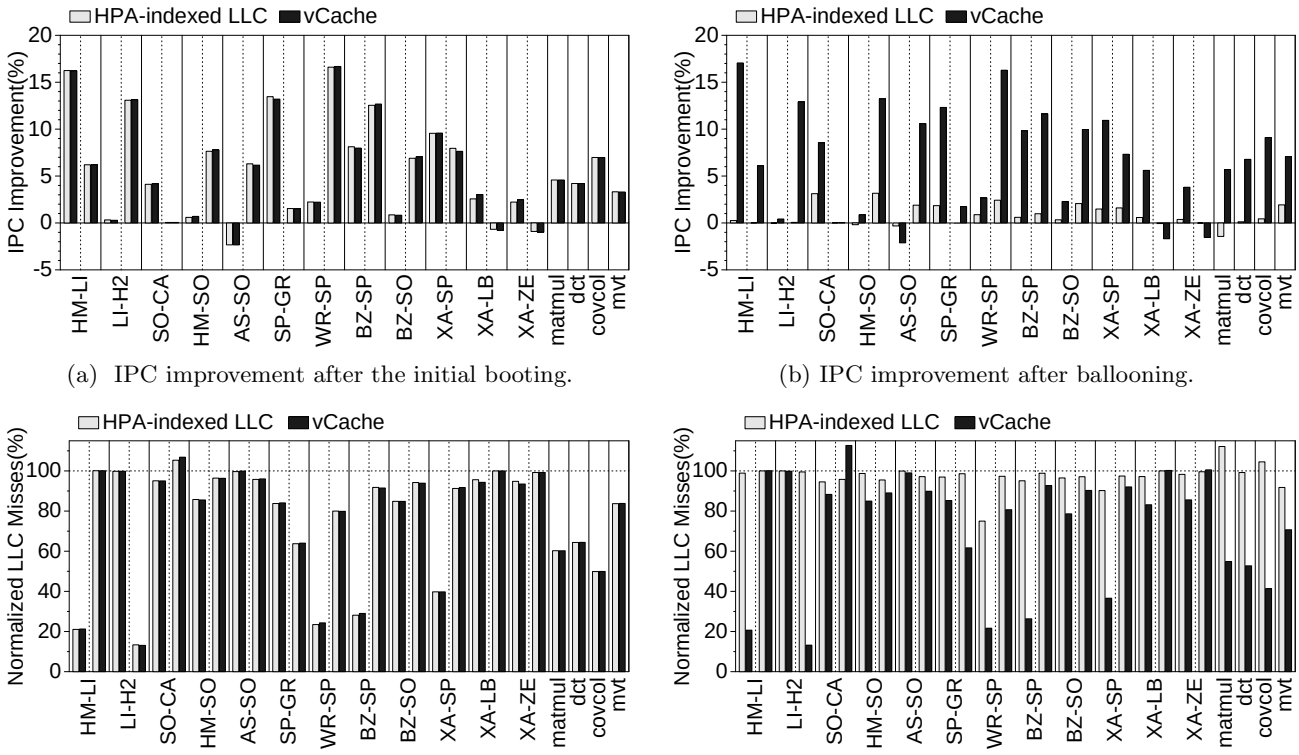
5.1 Transparency: Preserving Page Coloring

In this section, we demonstrate that *vCache* can provide a transparent vLLC for each VM, preserving the effectiveness of page coloring by a guest OS. We take a 4MB 8-way LLC for the pollute buffer mechanism and an 8MB 16-way LLC for ULCC to show the performance improvement with each page coloring mechanism; these two different LLC configurations are intentionally chosen for later demonstrations in which we consider a dual-VM system with two different LLC capacity allocations to two VMs. To demonstrate the preservation of page coloring, we evaluate two configurations (i.e., *traditional HPA-indexed LLC* and *vCache*) as a VM is initially booted and goes through a ballooning process. The initial booting has the GPA-to-HPA mappings close to an ideal case where the original page coloring by the OS is preserved, as described in Section 2.1. In contrast, the ballooning process quickly spreads memory pages of a GPA color across those of many HPA colors.

Figure 6 shows that *vCache* preserves the effectiveness of two page coloring schemes (i.e., pollute buffer and ULCC) in terms of normalized IPC and LLC misses. Figures 6a/6c and Figures 6b/6d represent the results for the initial booting and ballooning cases, respectively. All results are normalized to the baseline using a traditional HPA-indexed LLC without page coloring. The *y*-axis represents the normalized IPC improvement and the number of LLC misses in percentage. The first and second groups of two bars for each mix represent the results for the first and second benchmarks in the mix.

Although HPAs are used for indexing the traditional LLC, both page coloring schemes are still effective right after the initial booting, denoted by *HPA-indexed LLC* in Figures 6a and 6c. In all workloads, *HPA-indexed LLC* and *vCache* exhibit practically the same results in terms of IPC improvement and the number of LLC misses. The guest OS can obtain the effectiveness of its page coloring scheme even with a traditional HPA-indexed LLC right after the initial booting. After a few occurrences of ballooning, however, memory virtualization cannot preserve page coloring due to GPA-to-HPA mapping changes. In Figures 6b and 6d, ballooning makes both page coloring schemes ineffective for all workloads; we observe that the *HPA-indexed LLC* with the page coloring schemes does not improve the IPC and LLC miss rate compared to the baseline.

In contrast, regardless of GPA-to-HPA mapping changes after the ballooning process, *vCache* preserves the effectiveness of page coloring schemes by providing a transparent view of a vLLC to a VM. In Figure 6b, *vCache* shows 17% IPC improvement for *hmmer* in HM-LI by preserving the colors of all the pages except for VM-shared pages, while the *HPA-indexed LLC* shows less than 1% IPC improvement even with the page coloring scheme. Likewise, *vCache* shows 16% IPC improvement for *sphinx3* in WR-SP, whereas the HPA-indexed LLC with the page coloring schemes shows only 2% IPC improvement. Overall, *vCache* provides 6 percentage points higher geo-mean performance than *HPA-indexed LLC*. In terms of normalized LLC misses, *vCache* also preserves the effectiveness of the page coloring schemes for all workloads. In Figure 6d, we see that *h264ref* in LI-H2 shows a significant LLC miss reduction (87%), but the memory ballooning process completely eliminates the reduction in the *HPA-indexed LLC*. Overall, *vCache* offers 32 percentage points lower geo-mean LLC misses than *HPA-indexed LLC*.



(a) IPC improvement after the initial booting.

(b) IPC improvement after ballooning.

(c) Normalized LLC misses after the initial booting.

(d) Normalized LLC misses after ballooning.

Figure 6: Preserved effectiveness of guest operating system's page coloring (initial booting versus ballooning).

5.2 Isolation: Supporting Transparent vLLCs in Multi-VM Environments

In this section, we demonstrate that *vCache* can provide an isolated vLLC for each VM through a VM-based partitioning mechanism while preserving the transparency of each vLLC in a multi-VM system. As a multi-VM environment, we run two VMs concurrently, each of which adopts its own page coloring scheme (pollute buffer and ULCC). We choose eight SPEC CPU mixes for the pollute buffer mechanism in Table 1, and pair each mix with three PLUTO benchmarks (i.e., *matmul*, *dct*, and *covcol*). We exclude *mvt* since the execution time is very short compared with other applications. We model a fully shared 12MB 24-way LLC, assuming that the hypervisor allocates a 4MB 8-way and an 8MB 16-way vLLC to VM1 and VM2 with the pollute buffer and ULCC, respectively. Each VM runs on dedicated cores by pinning vCPUs, but shares the LLC.

To explore the impact of inter-VM interferences on the effectiveness of page coloring by each guest OS, we compare three configurations (i.e., *pLLC*, *vLLC*, and *sLLC*). For *pLLC*, we model two physically separated LLCs, and dedicate an LLC to each VM (4MB 8-way and 8MB 16-way), instead of a shared 12MB 24-way LLC when two VMs co-run. Thus, with *pLLC*, each VM solely uses a physically separated LLC, which prevents the other VM from interfering its LLC accesses. For *vLLC*, we isolate two vLLCs and assign a specified vLLC capacity to each VM through *vCache*. In our experiments, *vCache* provides a 4MB 8-way and an 8MB 16-way isolated vLLC for VM1 and VM2, respectively. For *sLLC*, we run two VMs on a 12MB 24-way LLC without using any partitioning scheme. *sLLC* shows the negative effects of inter-VM interferences experienced by the shared LLCs. Lastly, for all three configurations, we enable the pollute buffer mechanism and ULCC for VM1 and VM2, respectively.

Figure 7 shows IPC improvement of each application running on two VMs with the two page coloring schemes; VM1 runs SPEC CPU2006 mixes with the pollute buffer while VM2 runs PLUTO benchmarks with ULCC. For each application mix, the first two sets of bars show the performance improvements of two applications running on VM1, and the second set of bars show those of an application on VM2. The results are normalized to the baseline in which each VM solely uses its separate private LLC (i.e., *pLLC*) *with no GPA-based indexing and page coloring*. We mark the percentage numbers for the cases that show IPC improvement over 40%.

Overall, due to lack of LLC isolation between VMs and the resulted interferences, *sLLC* cannot preserve the effectiveness of page coloring by each guest OS unlike *vCache* enforcing the page colors through GPA-based indexing. More specifically, VM1 has a larger LLC footprint than allocated capacity and thus it can utilize more capacity in *sLLC*. Consequently, *sLLC*

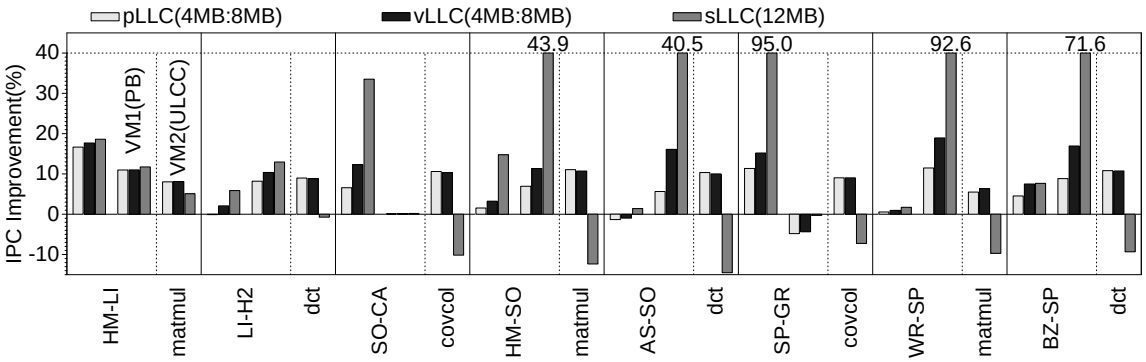


Figure 7: IPC improvement by guest operating system's page coloring with two consolidated VMs (VM1: pollute buffer, VM2: ULCC).

shows higher performance improvement than *pLLC* for VM1. However, VM1 nullifies the VM2's expected performance improvement with page coloring (i.e., ULCC) and thus the VM2's performance is significantly degraded even compared with the baseline. Although *sLLC* shows more IPC improvement (2% to 15% and 7% to 44%) than *pLLC* for HM-SO running on VM1, it shows even worse performance (11% to -12%) than *pLLC* for *matmul* running on VM2. In all combinations except for HM-LI and *matmul*, VM2 shows worse performance than the baseline.

In contrast, *vLLC* shows performance improvements close to or better than *pLLC* for both VMs. *vLLC* shows the better geo-mean IPC improvement by 2% percentage points than *pLLC*. Due to the work-conserving policy of *vCache*, VM1 can exploit unallocated/unused capacity along with its allocated vLLC and thus *vLLC* shows higher performance than *pLLC* while preserving VM2's performance. In other words, VM1 can exploit more capacity before VM2 fills its allocated capacity, since VM2 does not fully utilize its allocated capacity. Finally, Figure 7 demonstrates that the strong LLC isolation between VMs is necessary for *vCache* to preserve the effectiveness of page coloring as physically separated caches.

5.3 vLLC Capacity Allocation Policy

In this section, we explore the potential of *vCache* as a mechanism to support various vLLC capacity allocation policies that attempt to determine the optimal LLC capacity for each VM to improve the overall performance of a multi-VM system. Then we demonstrate that *vCache*'s ability to preserve page coloring of each guest OS plays a critical role to further improve the performance over hardware-based (dynamic) capacity allocation policies. To explore vLLC allocation policies, we evaluate four cases (i.e., *Best-Static-LLC*, *Dynamic-LLC*, *Best-Static-vCache*, and *Dynamic-vCache*) with two VMs, each of which runs its own page coloring scheme (i.e., pollute buffer and ULCC). The first two cases (i.e., *Best-Static* and *Dynamic*) employ a traditional HPA-indexed LLC and capacity allocation policies at VM granularity. For *Best-Static-LLC*, we find an optimal static capacity of each vLLC that maximizes

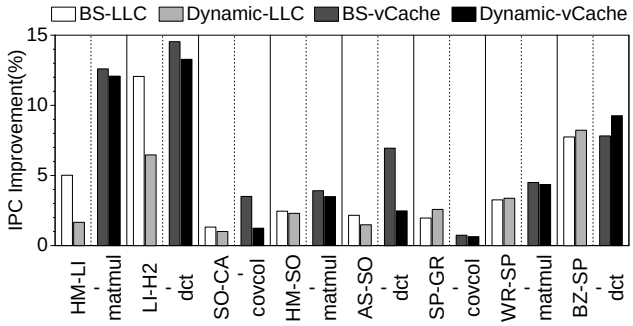


Figure 8: IPC improvement with capacity allocation policy (Best-Static (BS), Dynamic) : allocation policy only/allocation policy with *vCache* (geo-mean of three applications).

the overall throughput in terms of aggregate IPC values through an off-line exhaustive search approach sweeping the partition from 1:23 to 23:1 between two VMs. For *Dynamic-LLC*, we implement an allocation policy based on a utility-based cache partitioning (UCP) technique [25] – a dynamic way-based partitioning mechanism. Although the UCP tracks the LLC behavior associated with each core, for VM-based partitioning we track the LLC behavior associated with each VM instead of each core. *Best-Static-vCache* and *Dynamic-vCache* represent a GPA-indexed LLC preserving page coloring in addition to the same capacity allocation policies as *Best-Static-LLC* and *Dynamic-LLC*, respectively.

Figure 8 shows the improvement of aggregate IPC values of two VMs with a 12MB 24-way LLC. All results are normalized to the results of a traditional HPA-indexed LLC without VM-based partitioning and GPA-based indexing. Each bar represents the geo-mean of performance improvements from three applications running on two VMs. Although the page coloring mechanisms are not optimized for the allocation policies in general, *vCache* with capacity allocation policies shows further performance improvements than LLCs with the allocation policies at VM granularity without preserving page colors. While the capacity allocation policies improve the geo-mean performance for HM-LI and *mat-*

mul about 5% (*Best-Static-LLC*) and 2% (*Dynamic-LLC*), with the preservation of page coloring, *Best-Static-vCache* and *Dynamic-vCache* improve the geomean performance about 13% and 12%, respectively. For LI-H2 and *dct*, *vCache* with capacity allocation policies also shows further performance improvement than the HPA-indexed LLC with the allocation policies (*Best-Static*: 12% to 15%, *Dynamic*: 7% to 13%). By preserving the effectiveness of page coloring schemes, *vCache* has more opportunities for overall performance improvement.

6. RELATED WORK

Page coloring has been widely investigated in academic studies. The TLB slice [11] preserves cache behaviors of applications by simply inheriting the colors of virtual pages to physical pages. To reduce cache misses, some studies focus on even distribution of cache accesses across the whole cache capacity in a single application. [12, 13, 14, 15]. They reduce cache misses and balance cache accesses by controlling page colors. For multi-core processors, several studies [18, 16] proposed the mechanisms that partition the cache shared by multi-cores through assigning different page colors to each core or application. In this paper, as page coloring techniques for guest operating systems, we used pollute buffer mechanism [17] and ULCC [21] (explained in Section 2.1 and Section 5) amongst prior page coloring techniques. *vCache* preserves the effectiveness of the both page colorings by each guest OS, which can be nullified by memory virtualization in conventional systems.

Since dynamically re-coloring pages requires costly memory copies, there have been prior studies to reduce the re-coloring costs, thereby alleviating memory copies. Hot-page coloring [19] identifies frequently accessed pages as hot pages, and then re-colors only the hot pages instead of whole pages to reduce the number of memory copies. Awasthi et al. [20] proposed a mechanism that migrates pages without memory copies using a shadow address recorded in the unused bits of its page table entry, for non-uniform cache architecture (NUCA). By using the intermediate addresses, they can re-color pages to migrate them in NUCA instead of copying pages by changing physical addresses. In our approach, *vCache* is capable of zero-copy re-coloring for guest operating systems in a virtualized system. *vCache* uses GPA as a sort of intermediate address, and GPA-based indexing can potentially eliminate costly memory copies when a guest OS re-colors its pages, since *vCache* uses GPA to place the pages in an LLC.

For the management of shared caches in multi-core processors, several groups proposed policies for way-based partitioning between applications or cores. Suh et al. [31], Utility-based Cache Partitioning (UCP) [25] is a dynamic way-based partitioning mechanism by estimating application's cache utility to minimize the number of cache misses. Kim et al. [32] proposed the fairness metrics and evaluated policies using the metrics for way-based partitioning. Cook et al. [37] intensively evaluated way-based partitioning in software-based manners using real hardware supporting way-based partitioning. Virtual Private Caches (VPC) [33] is a hardware scheme for Quality of Service (QoS) in CMP-based multi-cores by partitioning cache capacity and bandwidth. They implemented a way-based partitioning scheme guaranteeing at least allocated shares (may have more) to each thread. *vCache* implemented a work-conserving partitioning policy similar to VPC, which improves LLC utilization with unused capacity. Although way-based partitioning has been widely studied, to our best knowledge, *vCache* is the first approach to implement VM-based partitioning at way-granularity while providing a transparent view of the shared LLC to each VM.

7. CONCLUSION

In this paper, we propose *vCache* that provides a transparent and isolated vLLC for guest VMs in virtualized systems. With little hardware overhead, *vCache* can facilitate various unchartered vLLC capacity-based services for the public clouds while improving performance compared to a traditional virtualized system. *vCache* first allows a guest OS to control the placement of memory pages in the LLC, as if the OS manages the LLC in a non-virtualized system, improving performance of a VM by up to 17% compared to a traditional virtualized system. Second, *vCache* can deliver almost the same performance as or even better performance than VMs with separate physical LLCs per VM through its vLLC isolation capability which is also critical to preserve vLLC transparency for each VM in a highly consolidated machine. Lastly, *vCache* can achieve further performance improvement with vLLC capacity allocation policies, increasing performance improvement of an existing capacity allocation policy from 2% to 12%, compared to a traditional HPA-indexed LLC without the policy.

Acknowledgment

This research was supported in part by a National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. NRF-2013R1A2A2A01015514), NSF grants (CNS-1217102 and CNS-1557244), and a DARPA grant (HR0011-12-2-0019). Nam Sung Kim has a financial interest in Samsung Electronics and AMD.

8. REFERENCES

[1] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer*, 1974.

[2] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat, "Enforcing performance isolation across virtual machines in Xen," in *Proceedings of the 2006 International Conference on Middleware*, 2006.

[3] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: Virtualized cloud infrastructure without the virtualization," in *Proceedings of the 34th International Symposium on Computer Architecture*, 2010.

[4] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware

clouds," in *Proceedings of the 5th European Conference on Computer Systems*, 2010.

[5] G. Somani and S. Chaudhary, "Application performance isolation in virtualization," in *Proceedings of the 2nd IEEE International Conference on Cloud Computing*, 2009.

[6] T. Cucinotta, D. Giani, D. Faggioli, and F. Checconi, "Providing performance guarantees to virtual machines using real-time scheduling," in *Proceedings of the 8th Workshop on Virtualization in High-Performance Cloud Computing*, 2010.

[7] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan, "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the 2011 USENIX Annual Technical Conference*, 2011.

[8] A. Gulati, A. Merchant, and P. J. Varman, "mclock: Handling throughput variability for hypervisor io scheduling," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, 2010.

[9] "Intel®Xeon®Processor E3-1220." [Online]. Available: `http://ark.intel.com/products/52269`.

[10] "Intel®Xeon®Processor E5-2699 v3." [Online]. Available: `http://ark.intel.com/products/81061`.

[11] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice-a low-cost high-speed address translation mechanism," in *Proceedings of the 17th International Symposium on Computer Architecture*, 1990.

[12] R. Kessler and M. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems*, 1992.

[13] E. Bugnion, J. Anderson, T. Mowry, M. Rosenblum, and M. Lam, "Compiler-directed page coloring for multiprocessors," in *Proceedings of the 7th Internatinoal Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[14] T. Romer, D. Lee, B. N. Bershad, and J. B. Chen, "Dynamic page mapping policies for cache conflict resolution on standard hardware," in *Proceedings of the 1st Symposium Operating Systems Design and Implementation*, 1994.

[15] 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.

[16] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared L2 caches on multicore systems in software," in *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[17] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer," in *Proceedings of the 41st International Symposium on Microarchitecture*, 2008.

[18] 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 *Proceedings of 14th International Symposium on High Performance Computer Architecture*, 2008.

[19] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th European Conference on Computer Systems*, 2009.

[20] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, 2009.

[21] X. Ding, K. Wang, and X. Zhang, "ULCC: a user-level facility for optimizing shared cache performance on multicores," in *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*, 2011.

[22] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[23] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[24] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd Symposium on Cloud Computing*, 2011.

[25] 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 International Symposium on Microarchitecture*, 2006.

[26] "vSphere." [Online]. Available: `http://www.vmware.com/products/vsphere`.

[27] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[29] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, 2002.

[30] M. Huang, M. Mehalel, R. Arvapalli, and S. He, "An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® Xeon® Processor E5 Family," *IEEE Journal of Solid-State Circuits*, 2013.

[31] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, p. 0117, IEEE Computer Society, 2002.

[32] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[33] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007.

[34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[35] S. J. Wilton and N. P. Jouppi, "CACTI: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, 1996.

[36] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral program optimization system," in *Proceedings of the 29th Conference on Programming Language Design and Implementation*, 2008.

[37] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proceedings of the 40th International Symposium on Computer Architecture*, 2013.