Appears in the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)

Virtual Snooping: Filtering Snoops in Virtualized Multi-cores

Daehoon Kim, Hwanju Kim, and Jaehyuk Huh
Computer Science Department, KAIST
{*daehoon, hjukim, and jhuh*}*@calab.kaist.ac.kr*

***Abstract*—Virtualization has been rapidly expanding its applications in numerous server and desktop environments to improve the utilization and manageability of physical systems. Such proliferation of virtualized systems opens a new opportunity to improve the scalability of future multi-core architectures. Among the scalability bottlenecks in multi-cores, cache coherence has been a critical problem. Although snoop-based protocols have been dominating commercial multi-core designs, it has been difficult to scale them for more cores, as snooping protocols require high network bandwidth and power consumption for snooping all the caches.**

In this paper, we propose a novel snoop-based cache coherence protocol, called virtual snooping, for virtualized multi-core architectures. Virtual snooping exploits memory isolation across virtual machines and prevents unnecessary snoop requests from crossing the virtual machine boundaries. Each virtual machine becomes a virtual snoop domain, consisting of a subset of the cores in a system. However, in real virtualized systems, virtual machines cannot partition the cores perfectly without any data sharing across the snoop partitions. This paper investigates three factors, which break the memory isolation among virtual machines: data sharing with a hypervisor, virtual machine relocation, and content-based data sharing. In this paper, we explore the design space of virtual snooping with experiments on real virtualized systems and approximate simulations. The results show that virtual snooping can reduce snoops significantly even if virtual machines migrate frequently. We also propose mechanisms to address content-based data sharing by exploiting its read-only property.

I. Introduction

Virtualization provides an illusion of multiple virtual machines on a physical system. A software layer called hypervisor manages physical resources and isolates virtual machines from each other. Virtualization has been rapidly expanding its applications in numerous server and desktop environments to improve the utilization and manageability of computing systems. In future many-core processors in virtualized systems, many virtual machines will be running on a processor, sharing cores dynamically.

The proliferation of virtualized systems opens a new opportunity to improve the scalability of multi-core architectures. One of the critical problems in multi-core designs is to reduce the costs of supporting shared memory through cache coherence mechanisms. Snoop-based coherence protocols have been dominating commercial multi-core designs, since they are simple and support fast cache-to-cache transfers. However, as the number of cores increases, the cost of broadcasting all requests will increase significantly, requiring higher network bandwidth, and consuming more power for looking up all the cache tags in a system.

In the context of non-virtualized systems, there have been several studies to reduce the overheads of snoop-based coherence by filtering out unnecessary snoop requests. Such filtering reduces power consumption for snoop tag lookups and network bandwidth consumption for broadcasting snoop requests [1]–[4]. The techniques maintain the sharing states of coarse-grained memory regions in hardware tables.

In this paper, we propose a snoop filtering technique, called *virtual snooping*. It reduces unnecessary snoop requests by dividing the cores in a virtualized system into virtual snoop domains. Since virtual snooping uses virtual machine (VM) boundaries to isolate snoop requests within a VM, it does not require any hardware table to track the private or shared states of memory blocks. A snoop request from a VM is sent only to the cores mapped to the VM. Using the existing VM boundaries, virtual snooping can be implemented on conventional snoop-based coherence protocols with a small cost. Optimizing cache hierarchy for virtual machines was first proposed by Marty and Hill, but it is based on two-level directory-based protocols [5]. To the best of our knowledge, this work is the first effort to improve snoop-based coherence for virtualized multi-cores.

In virtualized systems, each VM mostly accesses the physical memory pages allocated for the VM. For those *VM-private* page accesses, snoop requests do not need to be sent to the cores running the other VMs, which never keep the copies of the private cachelines of the requesting VM. Virtual snooping just needs to track on which cores a VM is currently running. Although it may be expected that most of the coherence transactions are for VM-private pages, there are three problems which break the isolation among VMs, and thus some cachelines used by a VM can exist in the caches of other VMs.

- **Data sharing with a hypervisor**: Although the majority of coherence transactions are for VM-private memory regions, cache coherence must still support communication through shared memory between VMs and a hypervisor. A hypervisor can run on any core in a system, and bring shared data to any cache. Virtual snooping cannot simply eliminate all coherence requests across VM domains to support such data sharing.

- **VM relocation**: The mapping between virtual CPUs (vCPUs) and physical cores is not fixed. A hypervisor relocates vCPUs to different cores to utilize physical cores as efficiently as possible. If a VM migrates to a new core, the cache of the old core will have the VM-private data of the relocated VM. Therefore, even if the VM is not running on the old core, the core must be in the snoop domain of the VM.
- **Content-based page sharing**: Another major source of data sharing across VMs is content-based page sharing among VMs. Sharing is used only for read-only pages, and once a VM updates a content-based shared page, a new physical page is allocated for the updated one by a copy-on-write mechanism. The content-based shared pages are guaranteed to be read-only and the external memory can provide the data even if cached copies exist in on-chip caches.

In this paper, we explore the design space of virtual snooping architecture considering the negative effect of the aforementioned three factors. A hypervisor can distinguish which pages are VM-private or shared across VMs. It records the sharing type of pages in shadow page tables or in guest-physical to host-physical mapping tables. Depending on page types, snoop requests are either broadcast to all the cores or multi-cast to only the cores used by the requesting VM.

The contributions of this paper are as follows. Firstly, using a real hardware system and the open-source Xen hypervisor, we present the effect of data sharing with a hypervisor. We show that memory accesses by the hypervisor account for a small portion of the total coherence transactions for most of our compute-intensive workloads. I/O-intensive workloads tend to have more memory accesses by the hypervisor, but the ratios are still under 20%. Secondly, we evaluate the effect of VM relocation on virtual snooping. Virtual snooping performs best when each VM runs on a fixed set of cores. However, pinning VMs to physical cores can lead to the under-utilization of cores. We propose a mechanism to track the number of the private cachelines of a VM in a cache. It uses per-VM cache residence counters in each cache. The mechanism allows virtual snooping to adjust the snoop domain of a VM dynamically, and thus prevents a VM from snooping unnecessary cores, when vCPUs migrate to different cores frequently. Thirdly, we evaluate the effect of content-based sharing on virtual snooping. For a subset of our benchmark applications, there are significant coherence transactions on content-shared pages. For such applications, broadcasting snoops on the content-shared pages reduces the effectiveness of virtual snooping significantly. We propose schemes to improve virtual snooping by exploiting the read-only property of content-shared pages.

The rest of the paper is organized as follows. Section II introduces the basics of virtualization and its implication on snoop filtering. In Section III, with a real virtualized system, we present the effect of memory accesses by a hypervisor and the frequency of VM relocation. In Section IV, we describe the architecture of virtual snooping, considering VM relocation and inter-VM memory sharing. In Section V, using simulations, we evaluate virtual snooping. In Section VI, we present the effect of content-based page sharing on virtual snooping and possible improvements. Section VII reviews related work and Section VIII concludes the paper.

II. Virtualization and Cache Coherence

A. The Basics of Virtualization

Machine virtualization enables multiple virtual machines (VMs), each of which encapsulates an individual computing environment, to efficiently share underlying hardware. In a virtualized environment, a hypervisor, a thin software layer directly located on hardware, fully controls the hardware resources and VMs. The primary goal of a hypervisor is to provide efficient resource sharing while ensuring performance and fault isolation. For CPU virtualization, the hypervisor allocates one or more virtual CPUs (vCPUs) to each VM. While a VM is provided with an illusion of its own dedicated cores, each vCPU is scheduled under the control of the hypervisor. As with general-purpose operating systems, the hypervisor scheduler aims at guaranteeing the fair allocation of CPUs among VMs.

Memory virtualization requires the hypervisor to maintain an additional page table for each VM to allow the isolated use of memory resources [6], [7]. The guest OS on a VM maintains per-process page tables which translate from guest-virtual addresses to guest-physical addresses. A VM is not permitted to access memory regions owned by others without necessary permissions. For isolated memory accesses, guest-physical addresses (virtual physical addresses) used by a VM are translated to host-physical addresses (real machine addresses) assigned by the hypervisor. Only the hypervisor is allowed to manipulate the guest-physical to host-physical address mapping. To make this translation efficient with TLBs, most hypervisors use software-based shadow paging [20], with which a hypervisor maintains direct page tables from guest-virtual to host-physical addresses. Recent processors support hardware-assisted address translation such as AMD nested paging [8] and Intel extended paging [9], in which hardware walkers traverse both guest page tables and hypervisor mapping tables for guest-physical to host-physical translation. For a para-virtualized OS, which is modified to be aware of the underlying virtualization layer, a hypervisor can maintain the guest-virtual to host-physical mapping directly in guest page tables [6]. For this direct paging, the hypervisor should validate every update of guest page tables to support VM isolation.

B. *Inter-VM Memory Sharing*

A hypervisor maintains a mapping between the guest-physical memory of a VM to the host-physical memory to provide the dedicated memory view for each VM. A VM does not usually access the memory area used by other VMs. Such memory isolation is the basis of virtual snooping. To filter coherence requests effectively across VMs, VMs must share as little memory as possible. Furthermore, vCPUs may migrate to different cores, which results in the data of migrated VMs left in the caches that other VMs are using. In this section, we discuss the two factors which reduce the effectiveness of virtual snooping: *inter-VM memory sharing* and *dynamic VM relocation*.

1) Inter-VM Memory Sharing: Although memory is mostly partitioned for each VM and a VM does not access the pages allocated to another VM, some memory regions can be shared among VMs for performance and efficiency. We explain three sources of memory sharing: *hypervisor*, *contents-based sharing*, and *inter-VM communication*. Firstly, the memory region of a hypervisor is globally shared among VMs. When a VM conducts a privileged operation, which must be validated and virtualized, its control is forwarded to the hypervisor. For this control, code and data of the hypervisor can be accessed from any VM. Accordingly, the shared hypervisor data can reside on the cache of any core.

Secondly, identical pages can be shared among VMs via an inter-VM memory sharing mechanism. Considering that memory is a limited resource for highly consolidated systems, the memory pages are shared by VMs based on the content of the pages. The content-based page sharing mechanism constructs the hashes of page contents to keep track of identical pages [10]–[12]. VMware ESX server is the first hypervisor that adopts this mechanism and reduces the memory footprints of homogeneous VMs by 10-40% [10]. A shared identical page can reside in the caches of different virtual machines.

Thirdly, direct inter-VM communications may use page sharing between the VMs. For example, to support high performance inter-VM networking, several techniques allow multiple VMs to communicate via inter-VM channels established on shared memory [13]. The inter-VM communication based on page sharing can improve the system throughput, when multiple VMs in a system intensively communicate with each other.

The hypervisor can identify all these inter-VM shared pages, and mark the page types in per-process shadow page tables or mapping tables for guest-physical to host-physical translation. Therefore, with a minor change in the TLB lookup mechanism, processors can know page sharing types for all memory accesses during address translation. Virtual snooping uses the sharing types to filter unnecessary snoops.

2) Dynamic VM Relocation: Besides explicit memory sharing, a hypervisor can relocate a VM via scheduling and management policies. From the view of CPU resources, VM relocation means the change of vCPU-to-core mapping. Dynamic relocation occurs when the hypervisor scheduler migrates a vCPU to another core for load balancing. To improve core utilization, the hypervisor scheduler attempts to make underlying cores as busy as possible. When a core is idle, the scheduler moves a vCPU that is waiting on a busy core to the idle core. Without dynamic scheduling, a core may become idle, while another core has many pending tasks. This relocation could make memory contents of a VM spread out across the caches of cores through which the VM passes.

Even though a VM allocates a fixed number of vCPUs, it may not always use all the allocated vCPUs, since the thread-level parallelism in the VM is dynamic and often there may not be enough threads to fully utilize the vCPUs. Such dynamic changes of active vCPUs makes loads on physical cores unbalanced. Furthermore, the hypervisor is capable of dynamically adjusting the number of vCPUs initially allocated to a VM. Most commodity operating systems support CPU hot-plugging to adjust available cores without downtime. This capability enables the hypervisor to deprive a VM of some vCPUs and re-allocate them to another VM, which requires more parallelism.

If VM migrations occur, VM-private data can reside in the caches of the cores on which the VM is not currently running. By VM relocation, a long-running VM may pass through all the cores in a system. It is possible that some VM-private data remain in all caches, making virtual snooping useless. Virtual snooping does not track each individual cacheline. Therefore, if there is a chance that a cacheline of a VM is in a cache, all snoop requests from the VM must be sent to the cache.

III. Measuring the Effect of Hypervisors

In this section, with experimental results from a real virtualized system, we present the effects of two factors which may reduce the benefit of virtual snooping. Firstly, we investigate L2 cache misses caused by a hypervisor and the privileged VM (domain0 in Xen). Those L2 misses must be broadcast to all the cores, even if virtual snooping is used. Secondly, we measure the frequency of VM relocation. VM relocation can make virtual snooping less effective, as the old core may contain the data of the previously located VM, and virtual snooping must send snoop requests to the old core too for the VM.

To measure the effect of a hypervisor, we use the Xen hypervisor (version 4.0) on a dual-socket system with two Intel quad-core x5550 processors. Each processor has four cores and each core has 32KB instruction L1 and data L1 caches, and a 256KB private L2 cache. The processor also has an 8MB L3 cache shared by all the cores on a chip. On each VM with four vCPUs, we run Linux 2.6.18.8-xen kernel. We evaluate PARSEC [14],

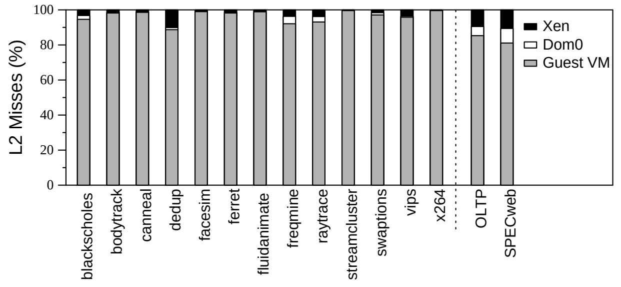


Figure 1. L2 miss decompositions: misses by hypervisor (Xen), domain0, and guest VMs

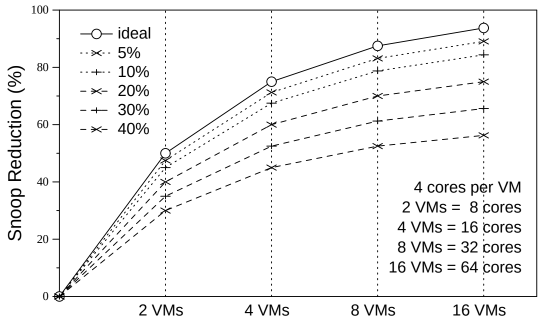


Figure 2. Potential snoop reductions with varying ratios of coherence transactions by a hypervisor to the total transactions: 2, 4, 8, and 16 VMs (4 cores per VM)

SPECweb2005, and OLTP as our benchmark applications. The OLTP application is from SysBench, and SPECweb2005 uses the SPECweb2005 Banking benchmark. To measure L2 misses by a hypervisor, we use a profiling tool (oprofile) which uses hardware performance counters. To measure the frequency of VM relocation, we use *xenperf* which shows the performance states of Xen including the number of migrations.

A. Cache Misses by a Hypervisor

Coherence transactions by a hypervisor must be broadcast to all the caches in a system, as the hypervisor can be invoked from any VM and its memory regions may reside in any cache in the system. Domain0 in Xen is a privileged VM which handles I/O for guest VMs. The hypervisor forwards I/O requests from guest VMs to domain0, and domain0 actually accesses I/O devices. Since domain0 serves all VMs, without explicitly pinning it to a specific core, it tends to migrate to different cores frequently. Although it is possible to reduce the set of cores domain0 can use, we allow domain0 to be scheduled to any core for performance. Due to the frequent relocation of domain0, from the perspective of virtual snooping, its effect is similar to that of the hypervisor. The coherence requests by domain0 must be broadcast too.

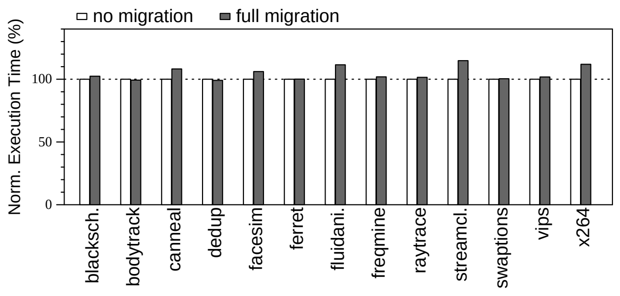
Figure 1 presents the decomposition of L2 misses by sharing types. For each result, we run two instances of the same application on two VMs. In the figure, L2 misses are decomposed to guest VM, domain0, and hypervisor. For PARSEC applications, less than 5% of L2 misses are for the hypervisor and domain0, except for dedup(11%), freqmine(8%), and raytrace(7%). For such compute-intensive applications, the ratio of coherence transactions which cannot be filtered by virtual snooping is low. For OLTP, 15% of L2 misses are for domain0 and the hypervisor, and must be broadcast. For SPECweb, 19% of L2 misses must be broadcast. Even for the I/O-intensive server workloads, L2 misses for domain0 and the hypervisor are less than 20%. Virtual snooping can potentially reduce snoops for more than 80% of L2 misses.

Figure 2 presents the potential reductions of snoops occurring in all the cores, compared to a broadcasting snoop-based protocol. For the figure, we assume VMs do not migrate, and the total number of virtual CPUs from all the VMs is the same as the number of physical cores. The number of virtual CPUs per VM is fixed to four. The x-axis shows increasing numbers of VMs and thus increasing numbers of physical cores. For example, the 4 VMs configuration has 16 virtual and physical cores. The figure shows six curves with different ratios of coherence transactions by a hypervisor to the total coherence transactions. As the number of VMs increases, and thus the ratio of the number of per-VM vCPUs to the number of physical cores decreases, virtual snooping can reduce more snoops. An ideal configuration with no hypervisor misses reduces more than 93% of snoops with 16 VMs running on 64 cores. As the misses by the hypervisor increase, the reductions decrease. However, with 5-10% hypervisor misses, the potential reductions are still 84-89% with 16 VMs.

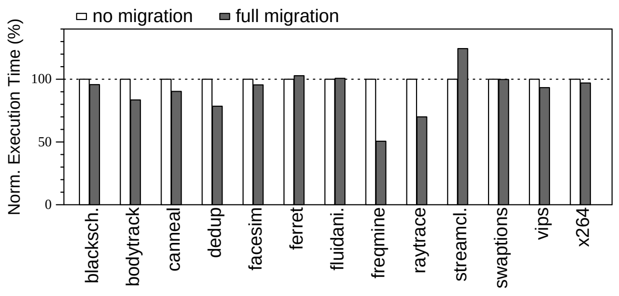
The simulator we use to evaluate virtual snooping later does not run a hypervisor and domain0 VM. Its result will present the snoop reductions only for coherence transactions by guest VMs. However, as shown in Figure 1, coherence transactions by the hypervisor and domain0 occur relatively infrequently, and thus, the lack of hypervisor activities in our simulator does not change the conclusion we draw from the restricted simulator.

B. VM Relocation

VM relocation moves vCPUs to different cores. A hypervisor makes the relocation decision based on its scheduling policy. The default scheduler of Xen is a credit-based scheduler, which is a proportional share scheduler with global load balancing on multi-core systems. The credit scheduler allocates a time slice to each vCPU, called credit, for each scheduling period. vCPUs consume the assigned credits as they run. For fairness guarantee, the scheduler always picks a vCPU that has remaining credits ahead of those that have run out of credits. Once a vCPU is picked, it can run for a time slice of 30ms. A vCPU can be blocked



(a) Undercommitted: two VMs (4 vCPUs per VM)



(b) Overcommitted: four VMs (4 vCPUs per VM)

Figure 3. The effect of pinning VMs: undercommitted vs overcommitted systems

Table I
AVERAGE VM RELOCATION PERIODS (MILLISECONDS)

Workloads	undercommit.	overcommit.
blockscholes	2880.6	91.3
bodytrack	26.1	1.2
canneal	28.4	3.4
dedup	10.8	0.1
facesim	30.0	1.2
ferret	375.9	31.5
fluidanimate	46.6	7.9
freqmine	1968.0	2064.4
raytrace	528.8	23.6
streamcluster	36.2	1.3
swaptions	2203.1	80.3
vips	18.3	0.7
x264	29.2	8.2
average	629.4	178.1

when it is no longer runnable, even if it has not used up the assigned credits.

For load balancing on multi-core systems, the credit scheduler dynamically relocates waiting vCPUs to idle cores. When all the vCPUs on a physical core have exhausted their time slices, the scheduler actively steals a waiting vCPU which has remaining credits from another busy core, and assign it to the idle core. This default scheduling policy does not consider the cost of migration. With this policy, all vCPUs aggressively migrate across physical cores to make cores as busy as possible.

An alternative way of scheduling to avoid migration is to pin vCPUs to physical cores. If a VM uses only a fixed subset of physical cores, the adverse effect by migration on virtual snooping can be reduced. However, such restriction on scheduling may result in under-utilization of cores. To show the effect of restricting physical cores a VM can use, Figure 3 presents execution times with different scheduling policies. The no migration policy pins virtual CPUs to physical cores with a one-to-one mapping. The full migration policy does not restrict migration to maximize the throughput of the system. The figure shows two results, one from an undercommitted system and the other from an overcommitted system. The hardware system has eight physical cores. Two VMs with four vCPUs per VM are running on the undercommitted system, and four VMs are running on the overcommitted system.

Figure 3(a) presents the normalized execution times when vCPUs are undercommitted. In the undercommitted system, pinning vCPUs to physical cores (no migration) results in better performance than the full migration policy by improving caching efficiency. However, as shown in Figure 3(b), in the overcommitted system, allowing migration provides much better performance than pinning vCPUs to physical cores. In the overcommitted system, improving the utilization of cores becomes critical, as multiple VMs compete for the cores. It is also possible to restrict the physical cores a VM can run to a subset of the cores in a system, instead of a one-to-one mapping. It will limit the size of the snoop domain of a VM, while it can reduce the load unbalance caused by the strict scheduling in the one-to-one pinning. Exploring such scheduling policies will be our future work.

Table I presents the average relocation period in milliseconds for any mapping changes between vCPUs and physical cores. For example, in blackscholes running on the overcommitted configuration, a vCPU changes its physical core every 91ms. The migrations of vCPUs of a VM within the current snoop domain of the VM do not negatively impact virtual snooping. However, the results in the figure include any mapping change for conservative evaluation. The overcommitted configuration shows much higher frequencies of migration than the undercommitted configuration does. In the worst case, the average period can be as short as 0.1ms. Also, the relocation periods vary widely depending on the behaviors of applications. Virtual snooping must be able to reduce snoops even with such frequent migrations.

IV. VIRTUAL SNOOPING ARCHITECTURE

A. Architecture

Exploiting isolation among VMs, virtual snooping sends snoop requests only to the cores (nodes) mapped to a VM. Although virtual snooping can be used with any cache hierarchy, in this paper, to simplify discussion, we assume a private L1 and L2 for each core. To identify the physical

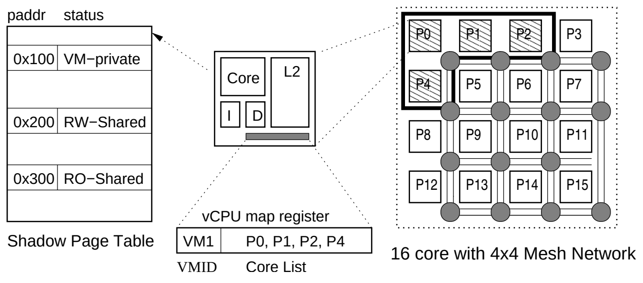
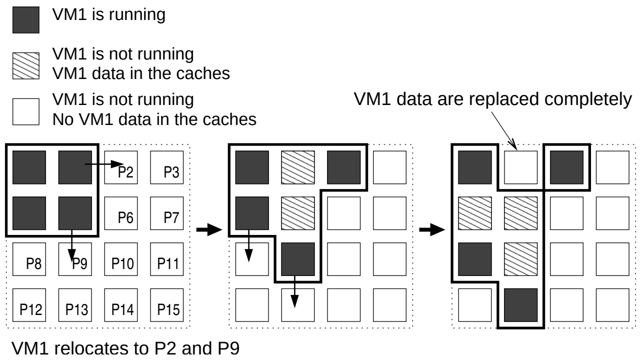


Figure 4. Virtual snooping architecture



VM1 is running
VM1 is not running
VM1 data in the caches
VM1 is not running
No VM1 data in the caches
VM1 data are replaced completely
VM1 relocates to P2 and P9

Figure 5. Virtual machine relocation

cores to which the virtual CPUs of a VM are mapped, each core has a register, called *vCPU map register*. The vCPU map register, n-bit vector for n cores, represents the physical cores used by the current VM running on a core. When a hypervisor schedules a vCPU of a VM to a physical core, the hypervisor is responsible to set the vCPU map to list all the physical cores the VM must snoop. The hypervisor must update the vCPU map before transferring the control to a VM.

Memory pages can be used by only a VM or shared among VMs and the hypervisor. Depending on the sharing types of pages, coherence requests are either multicast within a VM (by looking up the vCPU map), or broadcast to all the cores. The types of pages, VM-private or shared, are recorded in unused bits in page table entries. The type information is stored in per-process shadow page tables or nested page tables (guest-physical to host-physical mapping tables). In direct paging used by a para-virtualized hypervisor, the hypervisor sets the type information in guest page tables directly. These page tables are already required for virtualization, and virtual snooping needs only two unused bits in the page tables, which are available in most page table designs. For a coherence transaction for a VM-private page, the cache controller checks the vCPU map and sends snoop requests only to the cores listed in the vCPU map.

However, not all the memory used by a VM can be isolated within the VM boundary. There are two types of page sharing between a VM and the hypervisor or among VMs. The first type is page sharing to communicate data between a VM and the hypervisor, or between VMs (*RW-shared* pages). As discussed in Section II-B, this type of sharing results from hypervisor data accesses and inter-VM direct communications. Since the modified cache block can be in any core, snoop requests must always be broadcast for this type of shared pages. For example, some pages are shared between a VM and the hypervisor to communicate I/O requests and responses. For every page shared by the VM and hypervisor, the snoop requests must be broadcast, since the hypervisor can run and leave its data on any cache in the system. The second type is read-only page sharing to support content-based page sharing across VMs (*RO-shared pages*). We will discuss how to reduce snoops for content-shared pages in Section VI.

Depending on the types of memory pages (VM-private, RW-shared, RO-shared), virtual snooping must send snoop requests differently. The page sharing type bits (2 bits) must also be in the TLB to find the sharing type directly for every coherence transaction. Figure 4 presents the overall architecture of virtual snooping. Each core has a vCPU map register for the VM currently running on the core. The cores used by a VM must have the same vCPU map, which is maintained by the hypervisor. The shadow page table records sharing status, which can be updated only by the hypervisor.

B. *Supporting VM Relocation*

VM relocation may reduce filtering efficiency as the cache in the old location of an already relocated VM may contain some valid data of the VM. A naive method to guarantee the correctness is to flush the entire cache of the old node when a VM is relocated. However, flushing not only removes valuable data from on-chip caches causing expensive external memory accesses later, but also requires slow write-backs of modified blocks to the memory. If the VM-private data in the old location are not flushed, the vCPU map of the VM must include the old core, even if the VM is not running on the core. The old core cannot be removed from the vCPU map, since it may contain the data of the VM.

The vCPU map registers in the cores on which a VM is running, must be synchronized for each relocation. The hypervisor must update the vCPU map registers before relocation, and the hardware architecture can support a fast synchronization mechanism for vCPU map registers. The mechanism sends vCPU update messages with a new value to a subset of cores, and waits for acknowledgments from the cores. The latency is similar to or lower than broadcasting snoop requests and receiving responses. VM relocation is significantly infrequent compared to coherence transactions, and thus its cost is negligible. Figure 5 shows how the vCPU map is updated to include new cores, while the old cores are still in the map. In the second step of the figure, two new cores are added in the vCPU map of the VM. After adding the cores, the vCPU map has six cores, reducing the effectiveness of filtering by 50% from the optimal four cores.

However, although adding new cores to vCPU maps guarantees correctness, not removing obsolete cores from the vCPU maps may make virtual snooping useless eventually. As a VM migrates to different cores in a system, its vCPU map will include more cores, eventually containing all the cores in the system. Once it contains all the cores, the snoop filtering of virtual snooping does not occur. Unlike processes on conventional operating systems, which may finish its task and be destroyed after some time, VMs on the hypervisor tend to stay alive for a long time and thus eventually use all the cores in the system. Therefore, virtual snooping must support an efficient mechanism to remove obsolete cores from the vCPU map of a VM.

The first mechanism for efficient relocation support uses per-VM cache residence counters for each cache. Each per-VM counter records the number of VM-private blocks in the cache for a VM. Whenever a block is added to a cache, the corresponding counter for the current VM is increased. Cache tags must be extended to include a VM identifier for each block to mark the VM. When a cacheline is evicted by replacement or invalidated by snoops, the counter of the corresponding VM is decreased. When the counter becomes zero, it is certain that the private data of the VM do not exist in the cache, and then the core can be safely removed from the vCPU map of the VM. The core invokes the hypervisor to update the vCPU map registers of the removed VM. In the third step of Figure 5, a core is removed from the vCPU map, as it no longer contains any data for VM1.

A problem with the counter-based mechanism is that it waits for all cachelines for a VM to be evicted. However, it is possible that some data can stay in the cache for a long time, if a new VM running on the core has a small working set. Once the VM-private data of a VM spread to all the caches, the counter-based mechanism will not be able to filter any more snoop requests. A straightforward solution for the problem is to flush the cache selectively for a specific VM, if the counter is decreased under a threshold. However, it requires an additional controller logic to do it and it may need to check the entire cache to search the cachelines belonging to the VM.

In this paper, in addition to the counter mechanism, we evaluate a speculative mechanism to mitigate the effect of slowly evicted cachelines, using the property of the underlying Token Coherence protocol. In Token Coherence, if the first attempt of a coherence transaction fails for not being able to collect enough tokens, more transient request can be retried for the transaction. If the number of retries exceeds a threshold, Token Coherence resorts to heavy-weighted persistent requests which guarantee forward progress [15]. Using the property of Token Coherence, we remove a core aggressively from the vCPU map of a VM, even if the counter is not zero for the core. For a VM-private page, the first two attempts to collect tokens will send only to the cores in the vCPU map of the VM. If the two attempts fail, the next transient request will be broadcast. It removes a core from the vCPU map, when the VM is not running on the core and the cache residence counter becomes under a threshold (*counter-threshold* policy). The base counter mechanism can be used with any snoop-based coherence, but counter-threshold can be used only with coherence protocols supporting safe retries of coherence transactions.

Table II
SIMULATED SYSTEM CONFIGURATIONS

Parameter	Value
Processors	16 in-order SPARC core
L1 I/D cache	32KB, 4-way, 64B block, 2 cycle latency
L2 cache	256KB, 8-way, 64B block, 10 cycle latency
Coherence	Token Coherence, MOESI protocol
On-chip Network	4x4 2D mesh with 16B links
	4 cycle router pipeline

Table III
APPLICATION INPUT PARAMETERS

Application	Dataset	Application	Dataset
SPLASH-2		PARSEC	
cholesky	tk29.O	blackscholes	simmedium
fft	4,194,304 points	canneal	simsmall
lu	512 x 512	dedup	simsmall
ocean	258 x 258 grid	ferret	simsmall
radix	4,194,304 integers		
Servers			
SPECjbb2k	4 warehouses		

V. Experimental Results

A. Methodology

To evaluate virtual snooping, we use the Virtual-GEMS simulator [16]. Virtual-GEMS is based on Simics, a full system simulator, and GEMS, a timing simulator for memory hierarchies. Virtual-GEMS runs multiple instances of Simics to emulate virtual machines and feed the execution traces to the GEMS execution model. In this simulation environment, a hypervisor is not running, and its effect is not included. To overcome the weakness of this simulation methodology, we investigated the effect of a hypervisor separately using a real hardware system and the Xen hypervisor in Section III. Snoop reductions shown in this section is applicable only to the guest VM portions in Figure 1.

We model 16 in-order cores with 32KB L1 data and instruction caches, and a 256KB L2 private cache. The model uses Token Coherence for cache coherence among on-chip caches. The interconnection networks use the Garnet model for 4x4 2D mesh with 16B links. Table II shows

Table IV
NETWORK TRAFFIC REDUCTION OF VIRTUAL SNOOPING WITH IDEALLY PINNED VMS

Workloads	Reduction (%)	Workloads	Reduction (%)
cholesky	63.79	blackscholes	64.22
fft	63.20	canneal	63.35
lu	64.27	dedup	64.97
ocean	63.74	ferret	63.05
radix	63.39	specjbb	62.79
Average	63.68		

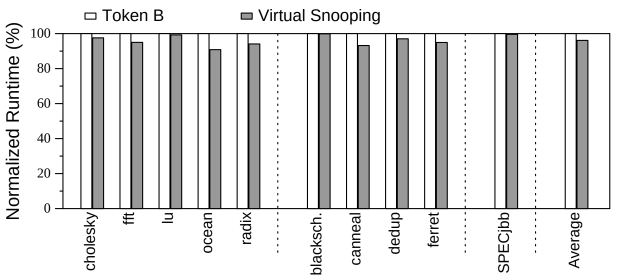


Figure 6. Execution times: virtual snooping with ideally pinned VMs

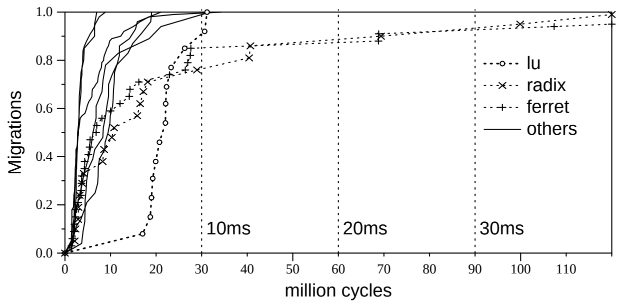


Figure 9. Cumulative distributions of the core removal period after a vCPU relocation in the counter mechanism (5ms migration period)

the detailed configurations for the simulated system. We run applications from SPLASH-2, PARSEC, and SPECjbb. The model uses four virtual machines with each VM having four vCPUs. The application input parameters, shown in Table III, are for a single virtual machine. For this paper, we do not model overcommitted systems due to the limitation of our simulator. The total number of vCPU is 16 (four for each VM), which is the same as the number of physical cores.

B. Ideally Pinned Virtual Machines

In this section, we evaluate the effectiveness of virtual snooping when VM migration does not occur. For this result, each VM is running on fixed four cores, and the vCPU-to-core mapping does not change. Without hypervisor or domain0 activities in Virtual-GEMS, all snoops are to VM-private pages, and thus no snoop requests must be broadcast. Therefore, in this ideal configuration, snoop reduction is always 75%, since a VM is using four cores out of the 16 cores in the system. The snoop reduction results in the reduction of not only snoop request messages but also power consumption for looking up cache tags. As discussed in Moshovos et al. [1], the snoop tag lookups will consume a significant portion of dynamic power of caches, as the number of caches increases.

Table IV presents the traffic reduction compared to the baseline TokenB, which always broadcasts requests. The measured network traffic is the total amount of data transferred through the network, including both data and coherence messages. Virtual snooping can reduce the total network traffic by 62-64%, compared to the broadcasting TokenB. Figure 6 shows the execution times normalized to the baseline TokenB. Virtual snooping in the ideal configuration reduces the execution time by 0.2-9.1%. The average execution time is reduced by 3.8%. The results show modest improvements in performance, as network bandwidth is not utilized intensively in this experimental configuration. This small improvement results are consistent with prior snoop filtering work [4]. However, the first goal of snoop filtering is to reduce the power consumption for snoop tag lookups and snoop message transfers, allowing the saved power budget to be used for other performance features or higher clock frequencies.

C. The Effect of VM Relocation

In this section, we evaluate the effect of VM relocation. However, without hypervisor activities, the Virtual-GEMS simulator cannot simulate the migration effect accurately. As an approximate method to simulate the migration effect, we shuffle the locations of two vCPUs periodically. Table I shows that the average migration period for an application in overcommitted systems can be as low as 0.1ms. Note that the migration periods in the table include all migrations both within and across VM boundaries. In this section, we simulate migrations only across VMs for conservative evaluation. We evaluate the migration effect with four different migration periods, 5ms, 2.5ms, 0.5ms, and 0.1ms. For example, for the 5ms configuration, two vCPUs from different VMs are randomly selected and their physical cores are exchanged every 10ms.

For the four migration periods, we evaluate three virtual snooping mechanisms. The base virtual snooping (vsnoop-base) does not remove cores from the vCPU map, as it does not check whether the migrated VM leaves any cachelines in the old location. The second mechanism (counter) has per-VM counters for each cache to count the number of VM-private cachelines in the cache. Once a counter reaches zero, the core is removed from the vCPU map of the VM. The third mechanism (counter-threshold) aggressively removes a core once the counter becomes less than a threshold. The threshold is set to 10 in the experiments, which is set to be low, not to remove cores from the vCPU maps prematurely.

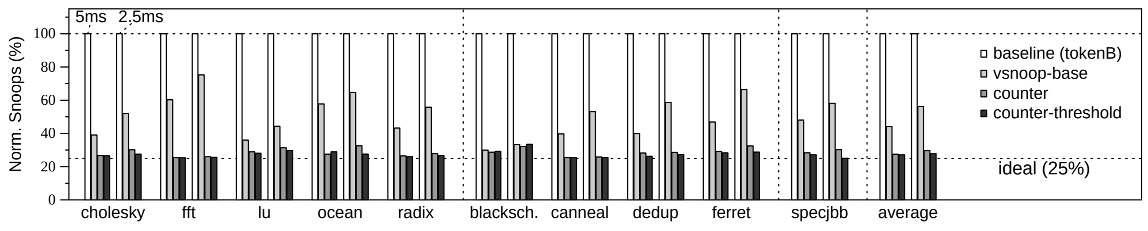


Figure 7. Total snoops with virtual snooping protocols: a vCPU is relocated every 5 or 2.5ms

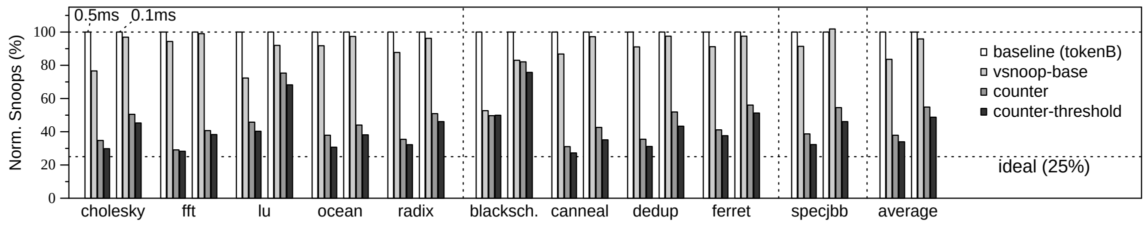


Figure 8. Total snoops with virtual snooping protocols: a vCPU is relocated every 0.5 or 0.1ms

Figures 7 and 8 present the total number of snoops occurring in all the cores, normalized to the baseline TokenB protocol. For each application, four sets of bars are shown for 5ms, 2.5ms, 0.5ms, and 0.1ms migration periods. Without counter-based optimizations, the base virtual snooping may lose its effectiveness significantly when VMs can migrate. As migration periods become shorter, more migrations occurs for each VM during its execution, and all the physical cores are included in the vCPU map of each VM very fast. With 0.1ms period, the base virtual snooping can only reduce 4% of snoops on average.

However, the counter mechanism can effectively remove obsolete cores from the vCPU map of a VM when the migration period is 5 or 2.5ms. The snoops with the counter mechanism is close to the ideal 25% from the baseline TokenB. With 0.5ms and 0.1ms, more vCPUs migrate to other cores, even before their previously located cores are removed from the vCPU map. The counter mechanism can still reduce 45% snoops on average even with a very aggresive 0.1ms migration period. Counter-threshold slightly improves the counter mechanism when the migration period is 0.5ms or 0.1ms. However, considering the restriction of the scheme requiring a fall-back mechanism in coherence protocols, its benefit seems to be too small to justify the additional complexity. Exploring more speculative schemes, which rely on the availability of safe retry in coherence protocols, will be our future work.

From the result, we observe that the VM-private data cached in the old location of a vCPU are replaced or invalidated relatively fast. Figure 9 presents the cumulative distributions of the periods to remove the previous core from the vCPU map of a relocated VM. The removal period measures the time difference from the relocation of a vCPU to the eviction of all the VM-private data of the VM from the old core. As shown in the figure, for most of the occurrences of vCPU relocation, the old core is removed from the vCPU map within 10ms. Two applications (radix and ferret) have a few occurrences of relocation with longer removal periods. For blackscholes which has short execution times and small working sets in L2 caches, the counters never become zero, and thus each relocation of a vCPU adds a new core to the vCPU map of the VM. In the results of blackscholes in Figures 7 and 8, the counter scheme does not improve the base virtual snooping, as it cannot remove any old core from vCPU maps. However, in the results, blackscholes shows snoop reductions even with vsnoop-base, since only a small number of migrations occurred during the short execution times.

VI. Virtual Snooping for Content-based Sharing

A. The Effect of Content-based Page Sharing

To consolidate multiple under-utilized systems to a virtualized one, the hypervisor allows the overcommitment of memory, and thus the total memory of all VMs running on the system can be greater than the actual physical memory. With such memory overcommitment, it is important to reduce the memory footprints of VMs to accommodate more VMs per physical system. Content-based page sharing allows read-only memory pages with the same content to be shared by VMs. The hypervisor maintains the hash values of all the memory pages for fast comparison, and checks their contents periodically. The content-based shared pages (content-shared pages) are marked as read-only pages, and any attempt to update them causes an exception, which invokes the hypervisor to create a copy of the page on write (copy-on-write mechanism). The content-based

Table V
THE PERCENTAGES OF L1 ACCESSES AND L2 MISSES FOR CONTENT-SHARED PAGES

Workloads	Access (%)	L2 miss (%)
cholesky	1.45	2.66
fft	5.43	30.64
lu	0.43	8.87
ocean	0.40	0.83
radix	20.47	0.96
blackscholes	46.16	41.10
canneal	25.16	51.49
ferret	3.64	5.13
SPECjbb	9.48	37.74
Average	12.51	19.94

page sharing, although it can reduce memory footprints significantly for certain workloads, can adversely affect virtual snooping, by increasing inter-VM memory sharing. Without any optimization for content-shared data, read requests to the content-shared pages must be broadcast to all the caches, as other VMs may use the pages simultaneously.

To investigate the effect of content-based sharing, we show how often coherence transactions occur on such content-shared pages. With the same configuration as described in Section V-A, we ran four VMs with the same application for each VM. The contents of all memory pages are compared, and the pages with the same content are marked as content-shared pages until one of the VMs attempts to update the pages. The content-shared pages include both application and guest operating system memory pages. Sharing detection in the experiment is more aggressive than what commercial hypervisors can do, since we ignore any performance impact of checking hash values of memory pages continuously. Therefore, the experiments show sharing statistics close to an ideal page-level content-based sharing mechanism.

Table V presents the ratios of the L1 accesses for content-shared pages to the total L1 accesses, and the ratios of the L2 misses for content-shared pages to the total L2 misses. As shown in the table, there are wide differences in the ratios of the L1 accesses and L2 misses for content-shared pages in different applications. Content-based sharing affects virtual snooping significantly only when a large number of L2 misses for content-shared pages occur. Among 9 applications, only 4 applications have more than 30% of L2 misses for content-shared pages. Without any optimization, coherence requests by those misses must be broadcast to all the caches.

B. Improving Virtual Snooping for Content-based Sharing

Exploiting the read-only property of content-shared pages, we can further reduce coherence requests on such pages. The hypervisor finds identical pages from different VMs and allows read-only page sharing among VMs. At the time when a page is marked as an RO-shared page, the hypervisor flushes any modified cachelines of the page to the memory to ensure the memory has a clean page. After flushing, there are no modified cachelines for the content-shared page, and any core or memory can provide the data for read requests. We propose three mechanisms to filter snoops on content-shared pages.

- *memory-direct* sends miss requests for content-shared pages directly to the memory, and the memory will provide the data. As used in CGCT [3], this technique can reduce snoops effectively on clean data. However, it may increase the latencies of L2 misses for content-shared data, since it does not check whether there are cached copies, which can be obtained by fast on-chip cache-to-cache transfers.
- *intra-VM* sends the requests only to the memory and the cores in the vCPU map of the requesting VM. If no cacheline is found in the intra-VM caches, the memory will send the data.
- *friend-VM* sends the requests to the requesting VM and another VM, *friend-VM*, which shares the most content-shared pages with the requesting VM. If the caches used by the two VMs do not have the data, the memory will provide the data.

To support the intra-VM and friend-VM schemes, the cache coherence protocol should be slightly modified. In common snoop-based coherence protocols, only one copy of a cacheline in the entire system is designated as a provider for the address. It prevents multiple cores from sending the same data to the requester. Even if several caches have shared copies of the same address, only one of the caches, which has the provider copy, must send the data. However, for the intra-VM and friend-VM schemes, virtual snooping

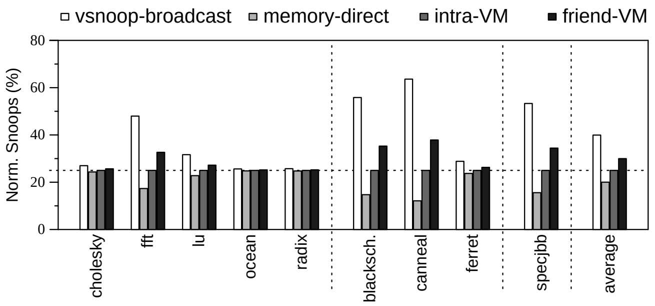


Figure 10. Expected snoops by memory-direct, intra-VM, and friend-VM optimizations

Table VI
POTENTIAL DATA HOLDERS FOR CONTENT-SHARED DATA

	fft	blacksch.	canneal	specjbb
Cache: all	47.3%	53.2%	63.9%	54.3%
Cache: intra-VM	0.1%	6.9%	26.9%	14.8%
Cache: friend-VM	24.4%	27.7%	21.0%	21.5%
Memory	52.7%	46.8%	37.1%	45.7%

must designate the provider copy for each VM, since read requests will go only to the intra-VM caches or friend-VM caches. The first copy of a cacheline brought into a VM can be the provider copy for the VM. For the friend-VM scheme, it is possible that two copies are transferred to the requester, as both the requesting VM and its friend VM may send a copy of the data.

Figure 10 presents the expected snoops occurring on every core, normalized to the baseline tokenB. The total snoops are estimated from the ratios of coherence transactions on content-shared pages to the total transactions and the required snoops for each scheme. Optimizations on content-shared pages affect four applications (fft, blackscholes, canneal, and specjbb). Memory-direct has the least snoops, often less than the ideal 25% snoops compared to TokenB, since it does not send snoop requests to the other cores for content-shared pages. All three optimizations can reduce snoops significantly compared to the base virtual snooping which broadcasts requests on content-shared pages. However, there are trade-offs between snoop reductions and L2 miss latencies in these three schemes.

Table VI presents the decompositions of data holders for L2 misses on content-shared pages. It first divides data holders into caches and memory. The external memory becomes a holder only if none of the on-chip caches have a copy of a missed address. For 37-53% of L2 misses on content-shared pages, the data holder is memory. However, for the rest of the misses, there is at least a cache holding the requested data, and the memory-direct scheme cannot use the cached copies. Although the memory-direct scheme has the most snoop reduction, it may degrade the overall performance by increasing miss latencies for 47-63% of the misses on content-shared pages.

When the data holder is a cache, it could be in a cache belonging to the requesting VM, a friend VM, or other VMs. The chances of getting data from the other caches in the requesting VM are relatively low from 0.1-27%. If the caches of a friend VM are included for snooping, the chances of cache-to-cache transfers increase significantly to 25-48%. The friend-VM scheme may reduce unnecessary external memory accesses significantly, reducing miss latencies for content-shared data. As the number of cores and the number of VMs increase, sending extra snoop requests to a similar VM will not increase the overall snoop count significantly.

VII. Related Work

There have been several recent studies to reduce the overheads of handling snoops for snoop-based coherence. The approaches mostly rely on tracking the sharing states of coarse-grained memory regions at requesting nodes, routers, and receiving nodes. RegionScount maintains region-based coherence filters at requesting nodes to avoid broadcasting snoop requests for private data [2]. Coarse-grain coherence tracking (CGCT) also uses additional coarse-grained coherence tags for each cache and tracks the private or shared states of regions, in addition to the conventional cacheline-unit coherence [3]. Snoop requests are either broadcast or sent directly to the memory depending on the coarse grain states. In-network Coherence Filtering (INCF) embeds region-based tracking in routers, and removes unnecessary snoops on the fly during the transmission of requests [4]. Instead of using additional filters or tags, Ekman et al. use TLBs to track the private or shared states of pages [17]. With operating system support, subspace snooping uses page tables to track sharers at page granularity [18]. Virtual tree coherence relies on the support from on-chip networks to build virtual multicasting trees. For each memory region, a virtual tree, which connects a subset of cores, is formed for snooping [19]. Destination-set prediction provides a speculative filtering mechanism [20]. Compared to the aforementioned studies, virtual snooping uses a virtual machine as a natural snoop domain, and also handles minor cases when data sharing crosses VM boundaries. Unlike the previous region-based filtering techniques which store sharing states or sharer information in on-chip storage or page tables, virtual snooping requires only a small addition in cache tags to add VM identifiers.

Marty and Hill (Virtual Hierarchies) inspired us to pursue this study [5]. They explore the design space of flexible cache hierarchies and two-level directory protocols for virtualized systems. Instead of designing a new protocol, virtual snooping uses a conventional snooping protocol, and improves the scalability of snooping protocols by filtering unnecessary snoops. Marty and Hill did not evaluate the effect of a hypervisor, VM migration, and content-based sharing. This paper looks into those issues which break the memory isolation among VMs. Rodrigo et al. proposed an efficient logic-based router design which can support network regions [21]. The network region can be used to divide cores into coherence domains in virtualized multi-cores.

VIII. Conclusions

In this paper, we proposed virtual snooping coherence, which can partition cores in a virtualized system into virtual snoop domains. It exploits memory isolation among VMs to remove unnecessary snoop requests. However, such memory isolation is not perfect in real virtualized systems due to data sharing with a hypervisor, VM relocation, and content-based data sharing. For data sharing due to a hypervisor, the results showed that coherence transactions from the hypervisor are relatively infrequent, mostly less than 10% of all transactions for the compute-intensive workloads, and less than 20% for the server workloads evaluated in this paper. For the VM relocation effect, we showed that a simple counter-based mechanism to check the cache residency of each VM, is effective enough to mitigate the impact of relocation. Finally, a subset of our benchmark applications have significant coherence transactions on content-shared pages. For such applications, virtual snooping must be further optimized to reduce snoops by exploiting the read-only property of content-shared pages.

With virtual snooping, future virtualized many-cores can be divided into small snoop domains. The snoop domains can dynamically change by hypervisor scheduling to maximize the system utilization. The number of cores is expected to continue to increase in future multi-core architectures. However, for many virtualized systems, VMs may use small scale virtual multiprocessors with a few vCPUs. As the ratio of the number of per-VM vCPUs to the total cores in a system decreases, virtual snooping will become more effective for scaling snoop-based coherence. Furthermore, the role of hypervisors will become critical to limit the size of snoop domains. It will be necessary to make hypervisors aware of the migration costs for virtual snooping. The hypervisors must limit the range of VM migration, as long as such restriction does not hurt the overall system throughput.

Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0025511)

References

[1] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary, "JETTY: Filtering snoops for reduced energy consumption in SMP servers," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, pp. 85–96.

[2] A. Moshovos, "RegionScout: Exploiting coarse grain sharing in snoop-based coherence," in *Proceedings of the 32nd International Symposium on Computer Architecture*, Jun. 2005, pp. 234–245.

[3] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Improving multiprocessor performance with coarse-grain coherence tracking," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 246–257.

[4] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network coherence filtering: Snoopy coherence without broadcasts," in *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, Dec. 2009, pp. 232–243.

[5] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 46–56.

[6] 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, pp. 164–177.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Transations on Computer Systems*, vol. 15, no. 4, pp. 412–447, 1997.

[8] "AMD64 virtualization codenamed pacifica technology: Secure virtual machine architecture reference manual," May 2005.

[9] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[10] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002, pp. 181–194.

[11] D. G. Murray, S. H, and M. A. Fetterman, "Satori: Enlightened page sharing," in *In Proceedings of the USENIX Annual Technical Conference*, 2009.

[12] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proceedings of the 8th Conference on Opearting Systems Design and Implementation*, 2008, pp. 309–322.

[13] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson, "Fido: Fast inter-virtual-machine communication for enterprise appliances," in *In Proceedings of the USENIX Annual Technical Conference*, 2009.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008, pp. 72–81.

[15] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness," in *Proceedings of the 30th International Symposium on Computer Architecture*, Jun. 2003, pp. 182–193.

[16] A. Garca-Guirado, R. Fernndez-Pascual, and J. M. Garca, "Virtual-GEMS: An infrastructure to simulate virtual machines," in *Proceedings of the 5th Int. Workshop on Modeling, Benchmarking and Simulation*, 2009.

[17] M. Ekman, P. Stenström, and F. Dahlgren, "TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors," in *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, New York, NY, USA, 2002, pp. 243–246.

[18] D. Kim, J. Ahn, J. Kim, and J. Huh, "Subspace snooping: Filtering snoops with operating system support," in *Proceedings of the The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2010, pp. 111–122.

[19] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti, "Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence," in *Proceedings of the 2008 41st International Symposium on Microarchitecture*, Washington, DC, USA, 2008, pp. 35–46.

[20] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared memory multiprocessors," in *Proceedings of the 30th Int. Symp. on Computer Architecture*, Jun. 2003, pp. 206–217.

[21] S. Rodrigo, J. Flich, J. Duato, and M. Hummel, "Efficient unicast and multicast support for cmps," in *Proceedings of the 41st Annual International Symposium on Microarchitecture*, 2008, pp. 364–375.