

# A Simulation Analysis of Shared TLBs with Tag Based Partitioning in Multicore Virtualized Environments

Girish Venkatasubramanian  
Renato J. Figueiredo  
Advanced Computing and Information Systems  
Laboratory  
University of Florida, Gainesville, Florida  
{girish, renato}@acis.ufl.edu

Ramesh Illikkal  
Donald Newell  
Intel Corporation  
Hillsboro, Oregon  
{ramesh.g.illikkal,  
donald.newell}@intel.com

## ABSTRACT

The current paradigm of computing in the server industry is undergoing rapid changes. Two of the most important changes are the emergence of multicore architectures with an increasing number of processors on a single die and the use of virtual machines (VMs) to efficiently partition and allocate these processors. As a result, the emphasis in micro-architecture design has shifted towards increasing the number of cores per die and providing hardware support for better performance of VMs on these cores. While this approach increases the utilization of shared resources in the processor, it renders the performance of one VM susceptible to the consumption of these resources by other VMs. Thus, there is a need to provide performance isolation and QoS guarantees for the VMs by managing the consumption of these shared resources at the VM granularity.

In this paper we present a mechanism for tagging the Translation Lookaside Buffer (TLB) entries and thereby partitioning the TLB space among different VMs. We focus on the TLB as improving the performance of the TLB and managing its use is very important, especially in virtualized environments. We use a combination of process specific and VM specific identifiers to tag the TLB entries, which results in improved TLB performance compared to using only VM-specific tags. We propose a hardware Tag Manager Table to efficiently generate and use these tags. By investigating the response of the VM to differences in the fraction of the TLB allocated to it, we show how this partitioning can be used for providing performance isolation and Quality of Service (QoS) guarantees for the VM.

## Keywords

Virtualization, TLB, Tags, QoS, Resource Management

## 1. INTRODUCTION

The current paradigm of computing in the server industry is undergoing rapid changes. This has resulted in two developments. The first is the emergence of chip multiprocessor architectures (CMP) where many independent cores are fabricated on a single die [19, 16]. Different threads (or processes) can execute concurrently on the different cores while sharing common resources. The second development is the increasing deployment of virtual machines (VMs) and the use of virtualization for server consolidation [9, 16]. Multiple workloads are classified into different classes, based on the nature of the jobs, their resource requirements or the service level agreements with the clients. Each of these classes is encapsulated in a virtual machine and many such VMs are deployed on a single powerful physical server. Due to this increased use of virtual machines, the X86 architecture has been evolving towards providing hardware support for improving the performance of VMs. Starting with the VT extensions in 2005 [25], there have been many changes in this direction including Intel VT for Connectivity and Intel VT Directed I/O [3]. Similar developments from AMD include the AMD-V virtualization technology and the Direct Connect Architecture [4, 5]. The latest change in this direction is the addition of tags as a part of the TLB entry and providing tag-comparison in hardware [17, 5, 4].

While this virtualization based consolidation model increases the utilization of shared resources, it also renders the performance of one VM vulnerable to how much of these shared resources are consumed by other VMs. Thus, there is a need for controlling and managing the usage of these resources by different VMs. This need for resource management becomes even more important to provide scalable and deterministic performance in future architectures like Datacenter-on-chip [12], where hundreds of heterogeneous services are consolidated on tens of cores. But traditional OS level management may not be a satisfactory solution in this highly virtualized environment [18]. Some of the resources, like the TLB on the X86 architecture, are not visible to the OS/hypervisor and thus cannot be managed by it. Moreover, many of these resources are designed to operate at extremely low latencies and the involvement of the OS or the hypervisor should be minimized. Any resource management policy which involves the OS or the hypervisor very frequently may cause a high latency and impact the performance of the workloads adversely. Hence, there is a need for implementing resource management mechanisms at the micro-architecture level with minimal involvement of the hy-

pervisor and no involvement of the OS running on the VMs and control resource usage of different VMs to be in-line with the system level QoS policies .

In this paper, we present such a mechanism for the partitioning and allocation of TLB space among different virtual machines by tagging the TLB entries with process specific identifiers and managing these tags using a Tag Manager. We focus on the x86 ISA and, aiming to support legacy guest O/Ss, we constrain our approach to a hardware-based TLB architecture, where the TLB management techniques commonly used in software managed TLBs like the SPARC and MIPS architectures cannot be implemented. While recent implementations of x86 processors have incorporated TLB tags, allowing some of the TLB management to be exposed to system software, there is a considerable legacy software base that has been written without support for tag management. For example, while a tag-aware hypervisor can tag virtual machine address spaces, a legacy guest O/S that is tag-unaware cannot benefit from this feature for processes within the VM. Requiring the hypervisor to be involved in tagging of guest address spaces is not desirable, because there are significant performance overheads associated with VM exits in guest address space modifications - hardware features such as extended page tables have been implemented to address this overhead. In this confine of hardware managed TLB with minimal hypervisor involvement, we propose an architecture for using a process-specific tag which can be derived from the value of the address space register, CR3. Using this architecture, different sized partitions of the TLB can be bound to different VMs ensuring performance isolation and satisfying QoS guarantees.

The remainder of the paper is organized as follows. An overview of the work related to our efforts is presented in Section 2. In Section 3, we give a brief description of the paging mechanism and the TLB in X86 and discuss the requirements for partitioning the TLB space. Then, in Section 4 we describe our proposed architecture for tag generation and for allocating TLB space amongst different VMs. Section 5 describes our experimental framework, the workloads being used and the metrics used to evaluate the proposed architecture. In Section 6 we briefly describe the results of our investigation and show how such partitioning can be employed for service-level QoS implementation.

## 2. RELATED WORK

Resource management in CMP platforms for providing QoS guarantees, especially the memory subsystem, has been the focus of many research efforts. Most proposals involve partitioning and sharing the last level cache. Kim, Chandra and Solihin [14] explore the sharing of caches with the aim of providing a fair share to threads of interest. Iyer et al [13] and Hsu et al [10] detail the different types of policies for sharing the last level cache, including policies for maximizing system throughput and for ensuring uniform throughput for each of the threads. Chang and Sohi [7] discuss adaptively increasing the allocation for a thread in the short run while maintaining fairness in the long run by explicitly scheduling which thread which get this increased share. Qureshi and Patt [20] base their scheme on deciding the cache allocation for a thread depending on how efficiently the thread uses the cache. Architectural support for OS level cache man-

agement, proposed by Rafique, Lim and Thottethodi [21], allows the flexibility of improved management policies at the OS level without large overheads.

TLB management by tagging the entries with address space identifiers is a well established technique in software managed TLBs like the SPARC [2] and MIPS [1] architectures. Such management has been shown to improve the TLB hit rate significantly. Sharing of the TLB space in X86, on the other hand, has not been exhaustively studied. One of the primary reasons is that the hardware managed TLB in the X86 architecture gets flushed every few thousand cycles causing the TLB entries to have a very short lifespan. Another reason is the inability to perform low latency tag-comparisons, which forms the basis of most of the cache-sharing schemes. Moreover, the delay caused by the TLB misses and TLB management may not be large enough in non-virtualized systems to require such allocations.

But, with the advent of virtualization, this is no longer true. TLB delays have been cited as a big barrier for improved VM performance [8]. This has led to the introduction of tags and a hardware-based tag-checking mechanism in the X86 architecture [17, 5, 4]. In AMD-SVM [4], each TLB entry has a 6 bit Address Space ID (ASID) as a part of its entry, uniquely identifying one of 64 address spaces to which it belongs. Currently, Xen on AMD-SVM [6] uses two ASIDs, 0 for the hypervisor and 1 for the guests. As long as the CPU is in Host mode, the TLB entries are tagged with ASID 0. When the CPU switches to Guest mode, the TLB is not flushed, but the ASID is changed from 0 to 1. Thus any TLB entry belonging to the hypervisor, but whose tag may match the virtual address of the guest, will not be declared a hit as the ASID tags will not match. Avoiding the TLB flush using ASID tags is found to have about 11% reduction in the overall runtime of the workload [6]. Similarly, in the Intel Nehalem [17], the TLB entries are tagged with a per-VM Virtual Processor Identifier (VPID). Tickoo et al [23] also explore TLB tagging in their Hypervisor Global approach where, the TLB entries belonging to the hypervisor, which are global within a domain<sup>1</sup>, are tagged with a VM specific identifier and not flushed during a switch between VMs.

Though, the primary intent of these efforts is to make the switching between VMs more efficient by avoiding a TLB flush, by using a process specific tag the TLB performance can be improved significantly. Moreover, the tagging mechanism may be used for partitioning the TLB space amongst different VMs and managing the TLB space that is allocated to each VM. In this paper, we propose a low-latency architecture for using a combination of process specific and VM specific identifier to tag the TLB entries. We discuss how the value of the CR3 register can be used to form this process specific identifier using a TLB Tag Manager. Such an approach significantly reduces TLB miss rates and the number of TLB flushes, compared to using only VM specific tags. We further propose using the TLB Tag Manager for partitioning the TLB space amongst many VMs for performance isolation and QoS mechanisms. Using our full-system experimental framework, we study the effect of using CR3 based TLB tags in a multi-processor environment with the

<sup>1</sup>In Xen terminology virtual machines are termed domains or doms

workloads on user domains (as is the case in a typical virtualized server setup) instead of running the workload on the control domain. Moreover, the workload we use is an OLTP class workload which spawns of multiple processes and has a different behavior compared to single-process CPU intensive SPEC workloads and is representative of the server environment.

### 3. REQUIREMENTS FOR PARTITIONING THE TLB

The X86 architectures is one of the most widely used architectures, especially for VM based server consolidation. In the X86 architecture, the linear address space seen by each process, called the virtual memory, is mapped into a smaller physical memory and disk storage [11]. Any address referred to by the process should be translated to a physical address, before it is used to access the memory. This mapping between virtual and the physical addresses is maintained in a set of multilevel hierarchical memory pages, termed as the page table. Every process has a unique set of page tables, identified by the Page Table Base address stored in the CR3 register. The process of looking up the page table to get a virtual address to physical address translation is termed a Page Walk. Page walking is a high latency process involving many memory reads. To reduce the overhead of the translations, the virtual address to physical address mappings are cached in the Translation Lookaside Buffer (TLB). The address translation process is also modified to look up the TLB before walking the page table. If the lookup misses in the TLB, the page walk determines the translation and enters it in the TLB. Moreover, since the entries in the TLB are valid only for the currently executing process, the TLB is flushed every time there is a context switch and a value is written into the CR3 register.<sup>2</sup> In this scenario, even if the TLB space is partitioned and assigned to two different processes, at a given time only the entries of one process will be cached. Any time there is a switch from one process to another, the entries of the first process will be flushed away. Thus the primary requirement for sharing the TLB space is to ensure that the TLB entries survive the context switch boundary, while ensuring that the entries belonging to the address space of one process are not accessed by utilized for another process. It should be noted that avoiding flushing the TLB during context switches is also the key for reducing the number of TLB misses.

One common solution which addresses this requirement is using tags as a part of the TLB entry which uniquely identifies the address space to which the entry belongs. The TLB lookup process should declare a match only if the tag of the current address space match the tag of the TLB entry in addition to other criteria. But, unlike the architected TLBs [22] where the OS has control over populating the TLB, the X86 TLB performs all TLB operations entirely in hardware to minimize the latency of the TLB management. Hence it is imperative that the overhead of any tagging mechanism is minimal.

Another important requirement for sharing the TLB is es-

<sup>2</sup>Though, it is possible to invalidate individual entries in the TLB, this "MOV CR3" method of flushing the TLB by writing a value in the CR3 is most commonly used.

tablishing discrete entities amongst which it should be shared. These entities are termed as service-classes, each identified by a service level tag SL. There are many ways to construct these different service classes - based on processes, based on the type of resources required or based on the service-level agreements with the different users of the system. With the advent of virtualization, virtual machines have become a favored method of establishing service-classes by encapsulating workloads in a virtual machine and specifying policies at the VM granularity. In this scenario it makes sense to develop QoS mechanisms which implement the sharing policies at the level of the virtual machine rather than at the thread level.

Having established these two primary requirements, we now discuss our proposed architecture for tagging and partitioning the TLB space.

### 4. PROPOSED ARCHITECTURE

As seen in the Xen port to AMD-SVM [6], currently the ASIDs are used to avoid flushing the TLB during a context switch from one VM to another, termed as an Inter-VM switch. But these tags do not prevent the TLB being flushed if there is a context switch between two processes within the same VM, an Intra-VM switch. By choosing tags that associate the TLB entries with a particular process address space rather than a particular VM, it will be possible to avoid both types of flushes and obtain higher hit rates for the TLB. But generation of this process-specific identifier should be performed without increasing the latency of the TLB lookup. This clearly rules out the Process Identifier as obtaining the PID requires interaction with the OS and is a high latency operation. However we capitalize on the fact that the CR3 contents are also unique per address space. Moreover, the contents of the CR3 can be obtained without querying the OS stack, as is the case with PID. Hence the TLB entries may be tagged with the CR3 to identify the process or virtual address space to which they belong. But the size of the CR3 register is quite large (32 or 64 bits). Such an addition in the TLB entry will increase the die area required for the TLB, the time for looking up the TLB and the energy expenditure for the TLB lookup.

We propose using a TLB Tag Manager Table (TMT) to handle this issue. The TMT is a small, fast, fully associative cache which is implemented at the same level as the TLB. Each entry in the TMT has three fields - a CR3 field which contains the CR3 value of a process, a Service Level (SL) field containing the priority class to which a process belongs and a VASI field which stores a unique identifier associated with the virtual address space of that process. The SL and the VASI, together constitute the CR3 Tag. Unlike the CR3 itself, this CR3 Tag is very small, ( $O(\log_2(N))$ ) where  $N$  is the number of entries in the TMT, resulting in a low-latency low-energy tag comparison process. In our experiments we use a 4-entry TMT with a 4 bit CR3 Tag comprising of a 2 bit VASI and 2 bit SL. This architecture is shown in Figure 1. It should be noted that the number of bits in the VASI limit the number of concurrent address spaces which can share the TLB. Thus the size of the VASI and the size of the TMT should be chosen with this in mind.

Using the TMT, most of the flushes arising from context

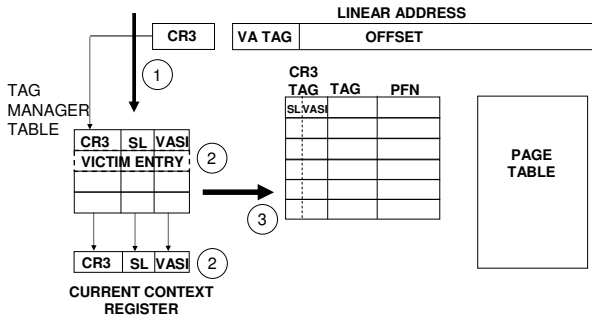


Figure 1: Flush behavior in a CR3 tagged TLB. In step 1 a value is written in CR3 prompting a flush. In step 2, the TMT is searched for the new CR3 and simultaneously the new CR3 is compared to the current CR3 in the CCR. The TLB and the TMT are flushed if the new CR3 matches with the CCR, or the new CR3 is inserted into the TMT evicting an entry as shown in step 3.

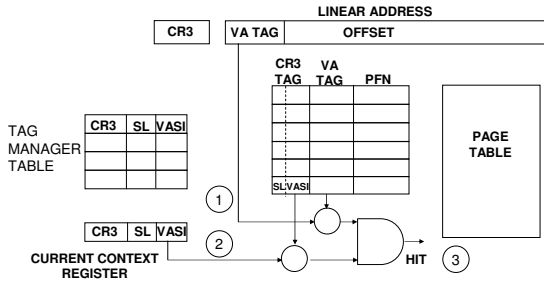


Figure 2: TLB Lookup when CR3 tags are used. In step 1 a possible match is found in the TLB, by comparing the tag of the virtual address with the TLB entries. In step 2 the tag from the TLB entry and the tag from the Current Context Register are compared. Only if both the CR3 based tags and the virtual address based tag match, the lookup results in a Hit as in step 3

switches can be avoided irrespective of whether they are inter-VM or intra-VM switches. Whenever there is a context switch from process  $P_1$  to  $P_2$ , the TMT is searched to determine if the CR3 value of  $P_2$  already exists. If it exists, that TMT entry is copied into the Current Context Register. The Current Context Register (CCR) is a register, with the same size as the Tag Manager Table entry, which caches the CR3, SL and the VASI for the current context. Using the CCR makes looking up the CR3 tag of the current address space very fast. On the other hand, if  $P_2$  is being scheduled for the very first time, its CR3 value will not be found in the TMT. Then the CR3 value is inserted into a free slot in the TMT and a VASI assigned to it. In both these cases the TLB is not flushed.

A situation may arise when, due to the limited capacity of the TMT, on a context switch from  $P_1$  to  $P_2$  where the CR3 of  $P_2$  is not present in the TMT, there are no free entries in

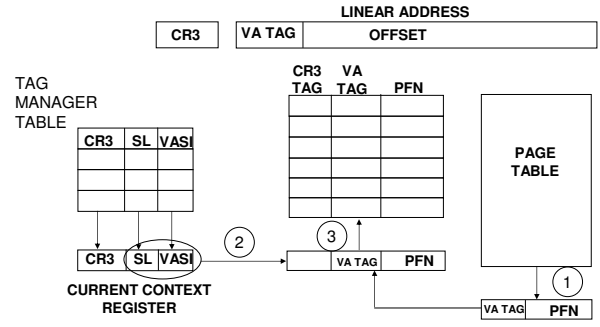


Figure 3: TLB behavior with CR3 based tags during a TLB Lookup miss. In Step 1, the physical address is determined by walking the page tables. In step 2, the TLB entry is composed by tagging the virtual to physical address mapping the CR3 tag from the Current Context Register. In Step 3, this augmented entry is cached in the TLB

the TMT. In this case, depending on the replacement policy, the victim entry ( $CR3_3, SL_3, VASI_3$ ), belonging to  $P_3$  is chosen. The CR3 and SL values of  $P_2$  replaces  $CR3_3$  and  $SL_3$  while  $VASI_3$  is reused for  $P_2$ . To avoid the TLB entries of  $P_3$  being used for  $P_2$ , the TLB is completely flushed. This flush, caused by the lack of capacity in the Tag Manager Table, is termed a Capacity Flush. The hypervisor may also update the page table for the current context. In this case the TLB has to be flushed to remove the stale entries. Such flushes are called Forced Flushes. The TLB Tag Manager detects these forced flushes by comparing the CR3 value from the CCR with the new value being written to the CR3 register every time the CR3 is being written. If both of them are the same, this rules out a context switch. This flush is detected as a Forced Flush and the TLB is flushed completely. Whenever the TLB is flushed, the TMT is also flushed to recycle the tags and free up space being occupied by contexts none of which have any entries in the TLB. This behavior is shown in Figure 1.

The TLB lookup happens as shown in Figure 2. The TLB is searched for any entry which has the same tag as the virtual address. Simultaneously, the VASI of the current context is looked up from the CCR. If a possible match is found in the TLB, the VASI of that entry is compared with the VASI from the CCR. Only if both match, the entry is valid and the TLB lookup results in a hit. If the lookup results in a miss, as shown in Figure 3, the page walk proceeds to determine the physical address from the page tables. Once this is determined, the entry is added in the TLB along with the CR3 tag (SL and VASI) of the current context.

The SL, as mentioned previously, is a service level tag which indicates the priority class to which the current process belongs. Since the allocation of virtual machines, we use the Virtual Machine Identifier itself as the SL. Multiple TMT entries may belong to the same virtual machine and will have the same SL. It should be noted that, though the SL forms a part of the CR3 Tag, it is not utilized in either the TLB

lookup or the TLB flushing on a context switch<sup>3</sup>. It is employed only when a new entry is added to the TLB. Whenever an entry belonging to a process in virtual machine  $VM_1$  is to be added in the TLB, the priority level  $SL_1$  of the new entry is obtained from the CCR. The victim is chosen by comparing the space occupied in the TLB by entries belonging to  $VM_1$ , i.e. entries with the SL part of their CR3 Tag set to  $SL_1$ , with the space allocated for that VM. For instance if the number of entries occupied by the current VM is greater than or equal to its allocation, and the replacement policy is LRU, the least recently used TLB entry with  $SL_1$  as the Service Level tag is chosen. On the other hand, if the current VM has not utilized all its allotted space, the victim is chosen from a domain which has exceeded its allotment.

While the architecture we describe is for a processor without virtualization support, it can be used for processors with Extended/Nested page tables (EPT/NPT) [17, 5, 4] with minor modifications. In a processor with EPT/NPT support, the regular X86 page tables (called guest page tables) are used to translate from the virtual address of a process inside a VM to the guest physical address of the VM. There is an additional set of Extended page tables identified by the EPT base register which are common for all processes in a VM and used to translate the guest physical address to the host physical address. In such processors the CR3 field of the TMT entry is augmented with the EPT base value. This augmented field, and therefore the CR3 tag, will still be unique per process address space.

## 5. EXPERIMENTAL FRAMEWORK

In this section we present an overview of the full-system simulation environment that we use to evaluate the impact of using CR3 Tags on the TLB performance and for QoS implementation. An overview of this simulation environment is shown in Figure 4.

Our experimental framework consists of Virtutech Simics (version 3.0.1). Simics is a full system simulation platform, capable of simulating high-end target systems with sufficient fidelity and speed to boot and run operating systems and workloads. It supports a variety of processor models like X86, X86-64, UltraSparc III and UltraSparc IV. The processor core of Simics is simulated at the functional level and does not include timing details. An atomic and sequential execution model is used wherein one instruction is fetched, decoded, executed and retired in one cycle, before the next instruction is fetched. Simics also provides the capability to install callback functions (also known as handlers, or HAPs) and associate these handlers with the occurrence of specific events both at the architectural level, like TLB misses, or in the code running on the architecture, like context switches. Simics also provides a configurable TLB model for the X86 processor consisting of four 64-entry (2 DTLBs and 2 ITLBs, one each for 4KB pages and one each for bigger pages), fully associative TLBs with First-In-First-Out as the replacement policy. We use this default model to simulate the TLB with no tags. We model the CR3 Tagged TLB by augmenting the

<sup>3</sup>In the following discussions the term CR3 Tag may refer to either the SL, if the context is QoS and performance isolation, or the VASI, if the context is TLB performance and avoiding flushes.

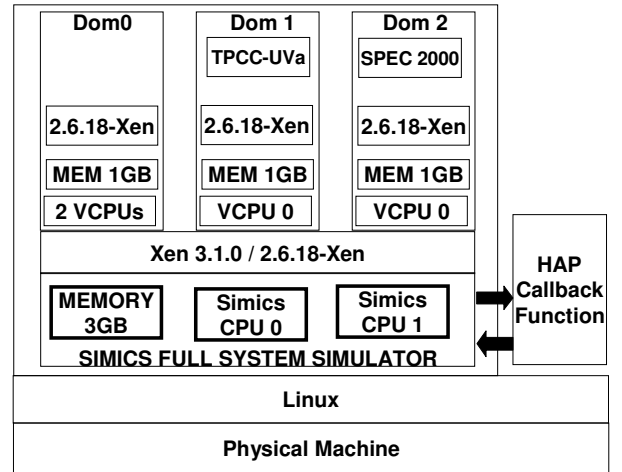


Figure 4: Experimental Framework

Simics TLB model with a Tag Manager, a CCR and adding CR3 Tags as part of the TLB entry. Using Simics, we simulate a machine with two P4 CPUs, 3GB memory and 20GB hard disk, as shown in Figure 4.

We have chosen Xen as the virtualization solution that we use. Xen is an open-source hypervisor which can support para-virtual guests running modified versions of operating systems (XenoLinux), or unmodified Hybrid Virtual Machines (if the processor has virtualization support built in). Due to its low overhead it has emerged as a popular choice for virtualization. On top of the Simics simulated "physical" machine, we boot Xen-3.1.0, which has Simics HAPs compiled in it to trigger various functions during inter-domain switches. On booting, Xen starts up a control VM or domain called dom0 with 2 VCPUs. From this domain user domains or domUs can be created. For this paper we have chosen Xen 3.1.0/2.6.18-xen kernel for both the dom0 as well as the guest domUs. Two guest domains, dom1 and dom2, are created, each with one VCPU and 1GB memory and equal weights for the scheduler.

The primary workload we use is an On Line Transaction Processing workload. OLTP class workloads are one of the typical workloads in server clusters. The TPC-C benchmark [24] is a well-know benchmark used to measure the performance of high-end server systems. We use the TPCC-UVa, an open source implementation of the TPC-C benchmark written by Llanos [15]. It uses the PostgreSQL database system and a simple transaction monitor to measure the performance of systems. As a secondary workload, we use vortex, a database manipulation workload, from the SPEC CPU suite of benchmarks. On dom1, we run the primary workload, TPCC-UVa and on dom2 we run the secondary workload which is either TPCC-UVa or Vortex. Thus we have two configurations, TPCC-TPCC (TPCC-UVa on both dom1 and dom2) and TPCC-Vortex (TPCC-UVa on dom1 and Vortex on dom2). Xen allows pinning a VCPU to physical CPU whereby that VCPU will be scheduled only on that physical CPU. In the previous two workloads, the VCPUs are not pinned to any physical CPU (so we call those workloads as TPCC-TPCC-

nopin and TPCC-Vortex-nopin). We also create two more configurations TPCC-TPCC-0012 and TPCC-Vortex-0012, where the two VCPUs of dom0 are pinned to physical CPU0 and the VCPU of dom1 and VCPU of dom2 are pinned to CPU1. By pinning the VCPUs in this manner, we can reduce the number of domain switches on CPU0, while the number of domain switches in CPU1 will be quite high. We can also study the effects of TLB allocation as dom1 and dom2 will be the only two sharing the TLB of CPU1 in the pinned workload configurations.

The primary metric measured by TPCC-UVa (similar to TPC-C) is the throughput which is defined as the number of New-Order transactions per minute a system generates while the system is executing four other transactions types (Payment, Order-Status, Delivery, Stock-Level). We have instrumented the source code of TPCC-UVa using Simics HAPs to keep track of the completion of every New-Order transaction. With this instrumentation it now becomes possible to account for per-transaction and per-instruction metrics. Moreover, using the callback function which is triggered whenever a domain switch occurs, we are able to maintain TLB statistics on a per-domain basis.

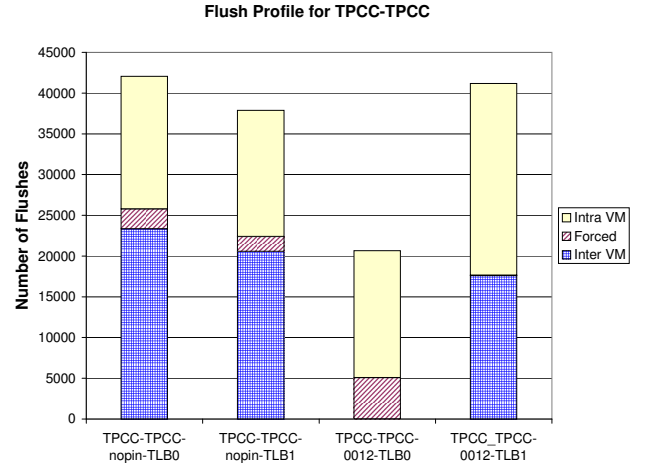
## 6. EXPERIMENTAL RESULTS

In this section we first evaluate the benefit of using CR3 tags over tagging with domain specific identifiers. Then we examine the reduction in the miss rate arising from using CR3 tags. We also investigate the variation of the TLB miss rate of a domain caused by varying its TLB allocation, and how this response differs with TLB size. We also look at the TLB miss rates over a period of time and discuss how a QoS policy can take advantage of the partitioning capability in the TLB to achieve its performance goals. In all the experiments performed, 10 billion instructions were retired on both CPU0 and CPU1 in the Simics simulated machine.

### 6.1 Comparing CR3 Tags with Domain Specific Tags

The benefit of using an identifier to tag the TLB and thereby avoiding flushes leads to a lower miss rate in the TLB and thereby a better performance. In the TLB where no tags are used, the hit rate in the TLB is limited, because of shortened life span of the entries. Generally, in the case of caches, if the size of the cache is unlimited, after the workload has been executing for some time, all the required data will be contained in the cache and the hit rate will asymptotically reach 100%. But this is not the case with TLBs. Even if the TLB size were unlimited, the hit rate would be limited due to the periodic purging of the TLB. Thus, if more flushes are avoided, the increased life-span of the TLB entry will result in higher hit rates.

TLB flushes can be classified into three types, based on the cause for the flush. The reason that the TLB has to be flushed can be either a context switch or that the page table has been modified, as described in Section 4. If the cause is a context switch, the two processes between which the switch happens could be within the same VM or could be part of different VMs. Based on this we classify a flush into three categories: flushes caused by a Intra-VM context



**Figure 5: Flush profile for two configurations running TPCC-UVa on both dom1 and dom2. In the pinned configuration, only dom1 and dom2 can be scheduled on CPU1. In the unpinned configuration there is no scheduling restriction.**

switch, flushes caused by a domain to domain or Inter-VM context switch and Forced flushes. We call this breakdown, the “flush profile”. The flush profile is a good indicator of the gain that can be achieved by using a tagged TLB. For instance, if the forced flushes dominate, then irrespective of whether a CR3 based tag is used or a domain specific tag is used, the TLB will still be flushed. Thus the number of flushes that can be avoided will be small, leading to smaller gains from using tagged TLBs.

We compared the flush profile for TPCC-TPCC-nopin and TPCC-TPCC-0012 workload configurations, as shown in Figure 5. When we look at the number of flushes in TLB0 in both the configurations, it becomes clear there is no inter-VM switch flush in the TPCC-TPCC-0012 case. This is because, due to the pinning, only dom0 can be scheduled on CPU0, thus there is no switching from dom0 to any other domain on CPU0. The total number of flushes on CPU0 in the nopin case is almost twice that of CPU0 in the pinned case. On the other hand, in the TPCC-TPCC-0012 configuration, both dom1 and dom2 share CPU1. Moreover, neither domain is idle and both have equal weights in the scheduling policy. Thus there is a periodic switching between both these domains on CPU1 leading to a large number of TLB flushes in TLB1. Another important observation is that, irrespective of pinning, the number of intra-VM flushes are quite significant. In fact these are the dominant cause of the flushes in the pinned case. Moreover the number of intra-VM flushes are not influenced by pinning as inter-VM flushes. This clearly shows the potential of using CR3 tags over VM specific tags, as the former can avoid both inter-VM and intra-VM flushes.

With this background, we look at the comparison of using CR3 tags to VM or domain specific tags, as shown in Figure 6. On the Y axis we plot the actual number of times that the TLB is flushed, combined for both CPU0 and CPU1 on

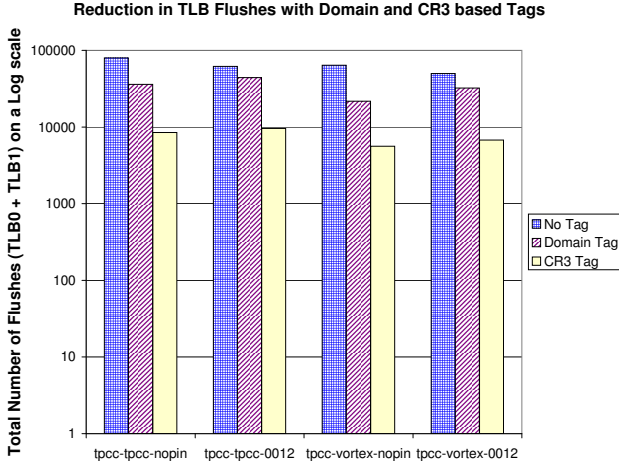


Figure 6: Comparing the number of TLB flushes using CR3 tags, domain specific tags and no tags. The Y axis is in Logarithmic scale and shows the total number of flushes for TLB0 and TLB1. Using CR3 based tags reduces the number of flushes by an order of magnitude, much more than using domain specific tags.

a logarithmic scale. From the figure, it is evident that, while using domain specific tags reduce the number of flushes, a much more significant reduction is obtained by using CR3 tags. In most cases, using domain tags reduce the number of flushes by 25% to 50%, as it can only avoid the flushes caused by inter-VM switches. Moreover, the performance benefit obtained by using such tags is highly dependent on the workload and whether the domains are pinned to CPUs. If, by pinning the domains to CPUs, as in both the TPCC-TPCC-0012 and TPCC-Vortex-0012 configurations, the number of inter-VM switches become quite small then the number of flushes avoided and the resulting performance benefit by using domain specific tags will be small. But using CR3 tags can avoid both Inter and Intra VM flushes. Apart from the forced flushes, the only other time the TLB is flushed is when any entry is evicted from the TMT. In our experiments, we used a four entry TMT, and obtained more than 90% reduction in the flushes. This reduction was the maximum in the TPCC-Vortex workloads. This can be understood by realizing that TPCC-UVa forks off many processes, each of which have their own address space. But vortex is a single process workload and thus takes up lesser number of entries in the TMT. Moreover, the number of flushes avoided can further be improved by having a bigger TMT, provided the latency for the TMT management does not add an overhead to the TLB operation.

## 6.2 Performance of CR3 Tags

In this section we investigate the reduction in the TLB miss rate as a result of tagging the TLB entries with CR3 tagging. The metric used for comparison is the number of TLB misses per ten thousand instructions retired - called MPTTI. We ran the TPCC-TPCC-nopin workload for 10B instructions on each core with TLB sizes ranging from 64 entries to 1024 entries and observed the change in MPTTI.

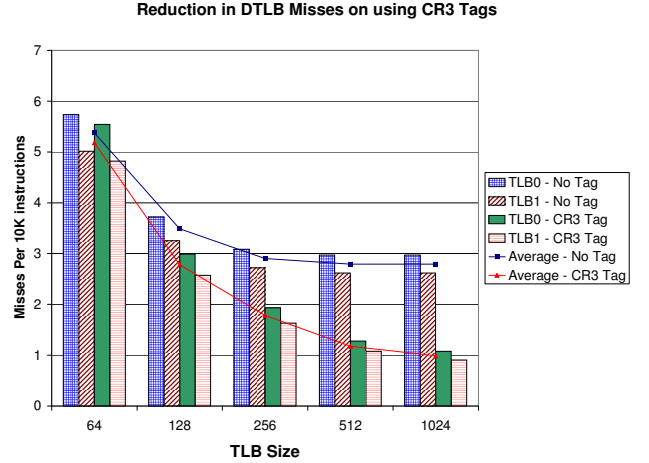


Figure 7: Reduction in the DTLB miss rate for TPCC-TPCC-nopin due to CR3 tags. The miss rates for TLB0, TLB1 and the average miss rate for both TLBs are shown. The improvement is small for a 64 entry TLB, but increases substantially for larger TLBs.

The reduction in DTLB MPTTI, shown in Figure 7, clearly indicates that TPCC is a workload which is TLB friendly, i.e., the miss rate decreases reasonably well on scaling the TLB size without any tagging. For instance, for the TPCC-TPCC workload, the MPTTI reduces from an average of 5.25 for a 64 entry TLB to 2.9 for a 256 entry TLB - almost a 50% reduction. But further scaling does not reduce the miss rate as seen from the flattening of the average MPTTI curve. This is due to the fact that most capacity misses have been satisfied and the dominant cause for the remaining misses are the TLB flushes. But when the CR3 tags are used, the miss rate can be further reduced. At 64 entries, the use of CR3 tags does not show significant improvement, and reduces the MPTTI by only 0.1, a small 2%. As the TLB size increases, the reduction of TLB misses due to CR3 tagging increases and overshadows the TLB scaling effect at sizes around 256 entries. For a 1024 entry TLB the MPTTI is around about 3 times smaller than the un-tagged 1024 entry TLB, and less than 20% of MPTTI of the 64 entry TLB without tags.

A similar behavior can be observed in the ITLB miss rates, shown in Figure 8. The ITLB miss rates are generally lower than the DTLB values, because of smaller footprint of the memory pages accessed for fetching the instructions and the increased locality of these pages compared to the data pages. Hence the improvement due to CR3, even at a size of 64 entries, is more pronounced in the ITLB (about 8%) compared to the DTLB. At sizes of 1024 entries, the MPTTI with tagging is only 0.48, which is a five-fold reduction compared to the 64 entry TLB without any tagging. It can also be seen that the slope of the average MPTTI curve is smaller for the ITLB than the DTLB, especially at sizes larger than 512 entries. This is due to the fact that, even with multiple processes sharing the ITLB, a capacity of 512 or 1024 entries begins to accommodate the entries of all the processes.

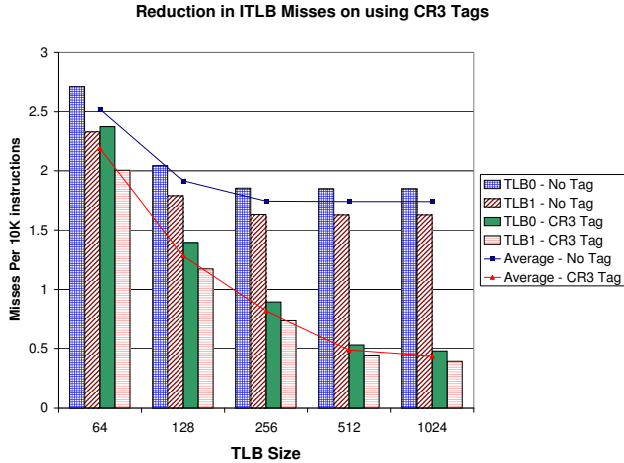


Figure 8: Reduction in the ITLB miss rate for TPCC-TPCC-nopin due to CR3 tags. The miss rates for TLB0, TLB1 and the average miss rate for both TLBs are shown. The improvement is significant even for a 64 entry TLB.

From these experiments, it can be observed that the CR3 tags improve the MPTTI, even at small TLB sizes. But the bigger payoff is at sizes beyond 256 entries. With the advent of multi-level TLB hierarchies, 256 or 512 entry last level TLBs are becoming a viable architectural design. Depending on the actual size of the different levels of the TLB, a scheme where CR3 tagging is used only at the larger last level TLBs may be envisioned.

### 6.3 TLB Partitioning and Allocation

One of the advantages of our Tag Manager architecture is the using the tags to associate different entries in the TLB with different processes belonging to different priority levels. As described in Section 4 we use the domain ID as the SL part of the CR3 tag. Whenever a new entry is to be added, we choose the victim based on the SL of the new entry and how many entries with the same SL already exist in the TLB. We run the TPCC-TPCC-0012 workload to evaluate how efficient is such a partitioning of the TLB space in controlling the TLB miss rate, which in turn influences the performance of the workload running in the VM. We choose to study the pinned workload configuration as pinning ensures that TLB1 is shared only by dom1 and dom2, making it possible to study the effect of the TLB usage of dom2 on the performance of dom1 without the interference of the third domain. The TLB of CPU0 is not shared, as only dom0 can be scheduled on CPU0. But the TLB in CPU1 is shared among dom1 and dom2 in varying proportions, all the way from a 10% to 90% share for dom1. We plot the miss rates for the DTLB of CPU1, both on a per-domain basis and averaged among both the domains, in Figure 9.

From this figure, it is quite apparent that the allocation of the TLB space serves as good control knob for varying the MPTTI, and thereby the overall performance of the TLB, at smaller TLB sizes. Looking at the family of curves for the 64 entry TLB, we see that the maximum average through-

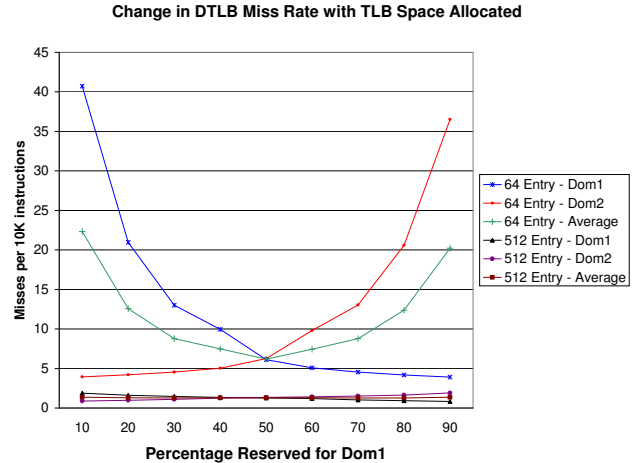


Figure 9: Response of the CR3 Tagged TLB miss rate for TPCC-TPCC-0012 to varying allocation of the TLB space for each domain. The DTLB miss rates for dom1, dom2 and the average miss rate for both domains combined are shown for two different TLB sizes. The miss rates show a strong correlation with their allotments.

put from CPU1, corresponding to the minimum average MPTTI, is obtained at 50%, i.e. when dom1 and dom2 have equal share of the TLB. This is intuitive considering the fact that the workload running on both the domains is TPCC-UVa, and so both domains exhibit similar TLB behavior. Moreover, this behavior is dependent on the number of domains sharing the TLB and how memory intensive these are. For instance, at a 512 size TLB, even a small fraction of the TLB translates into enough number of entries to satisfy the space demands of the workload. Hence, there is only a small variation in MPTTI when dom1's allocation is varied.

### 6.4 Performance Isolation by Partitioning the TLB

In the previous section we explored the issue of how changing the TLB allocation for a domain changed its miss rate. In this section, we investigate the flip side of the issue, i.e. how susceptible is the miss rate of a domain to the influence of other domains which share the TLB. We looked at the MPTTI for dom1 in two cases, namely TPCC-TPCC-0012 and TPCC-Vortex-0012 for a 64 entry TLB. The only difference in both these cases was the behavior of the dom2 workload. The results are shown in Figure 10.

From the figure it becomes clear that partitioning the TLB space based on CR3 tags provides good performance isolation. It can be seen that, irrespective of the TLB miss rate of dom2, the MPTTI for dom1 depends only on the fraction of the TLB allocated to it. From observing the MPTTI for dom2 in the TPCC-Vortex-0012 case, we realize that Vortex is highly TLB hungry. Whereas in the TPCC-TPCC-0012 case, TPCC-UVa on dom2 needs less TLB space compared to Vortex and its MPTTI is much smaller than Vortex. Clearly, the behavior of both these workloads is different. But, in both these cases the miss rate for TPCC-

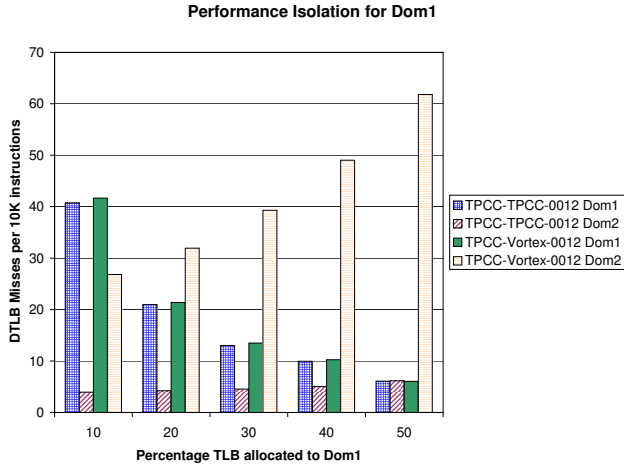


Figure 10: Performance Isolation using CR3 tags. The miss rates for dom1 and dom2 are shown for two different workload configurations. Despite the highly different demands on the TLB by dom2 the miss rate of dom1 is isolated from the influence of dom2.

UVa on dom1 is the same as long as its share of the TLB is maintained. The difference between the MPTTI for dom1 for TPCC-TPCC and TPCC-Vortex configurations has an average value of 2.2% and a maximum value of 3.6% for 30% allocation. This clearly indicates an effective isolation of the miss rates, and thereby the performance, of dom1 from the influence of other domains.

## 6.5 Towards Autonomic Control of TLB partitions

In Section 6.3, it could be seen that the optimum performance was obtained when both dom1 and dom2 have an equal share of the TLB. This was due to the fact that both the domains run the same TPCC-UVa workload and exhibit similar TLB behavior. But it is important to realize that, while the similarity of TLB behavior is true in the long run, each domain may exhibit varying phases with different TLB utilization in the short run. To study this, we obtained the MPTTI for dom1, dom2 and the average MPTTI for CPU1 at different time intervals, as shown in Figure 11. The interval boundaries are aligned with the flushing of CPU1's TLB.

From the Figure 11, we can identify different phases in the TLB behavior for both the domains. In phase 1, the interval between  $1.01095 \times 1E^{11}$  and  $1.01115 \times 1E^{11}$  cycles. In this phase, when the TLB space is split up with 30% for dom1 and 70% for dom2, the miss rate of dom1 is higher than the miss rate of dom2. But when the TLB partition is changed to award an equal share to both the domains, their miss rates become almost the same. By taking away 20% TLB space from dom2 to dom1, the miss rate of dom1 is lowered more than the increase in dom2's miss rate. If the QoS policy is to ensure minimum miss rate for dom1 or to ensure minimum average miss rate, then awarding 50% to dom1 is a good choice.

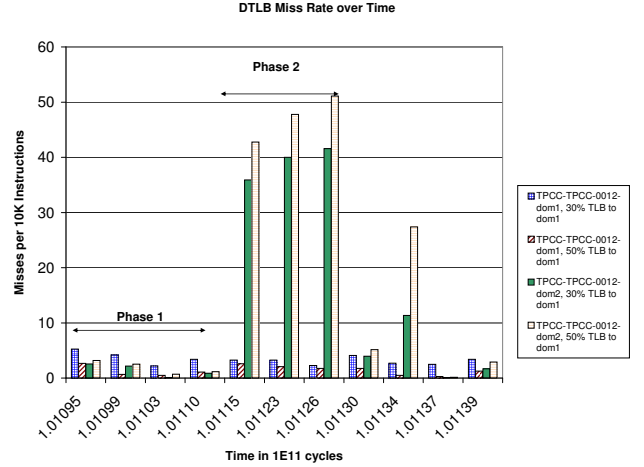


Figure 11: The TLB demands by the different domains have different phases. An autonomic QoS policy will be able to exploit these differences for enforcing the QoS policy

But in phase 2,  $1.01115 \times 1E^{11}$  cycles to  $1.01130 \times 1E^{11}$  cycles, dom2 exerts a much higher demand on the TLB than dom1. If we allot equal shares to both the domains, the average miss rate is much higher than if we allot 30% to dom1 and 70% to dom2. In this phase, if the QoS policy is to ensure maximum performance for dom1, it should be allotted 50% of the TLB. But, if the QoS policy is to minimize the average miss rate, then the allocation should be changed to award 70% to dom2 and 30% to dom1.

From the above discussion, it follows that a static allocation of the TLB space will not suffice, as the pressure exerted on the TLB by different domains varies with time and with the type of workload running on the domains. Moreover a change in the priority of the domains will necessitate a re-allocation of the TLB. Thus there is a clear need for a QoS system which takes into account all these factors and allocates the TLB space autonomically, without the need for external intervention. It should be noted that the TLB flushes form a natural point in time when such a policy can change the reservations.

## 7. CONCLUSION

In this paper we propose the use of tags as a means of increasing the TLB hit rate and for partitioning and managing the allocation of the TLB space among various VMs in the X86 architecture. The novelty in our approach is the use of tags comprising of process-specific identifiers, to improve TLB performance and VM-specific identifiers for QoS management. The process specific identifier is inferred from the hardware address space pointer (CR3 register) and a Tag Manager Table in conjunction with the hardware based tag-comparison to implement this tagging with low latency. Using CR3 Tags reduces the number of TLB flushes by an order of magnitude compared to using no tags and by 4 times compared to using only domain specific tags. The TLB miss rates can also be reduced by about 3 to 4 times when CR3 tags are used.

We also propose using the CR3 tags for allocating fractions of the TLBs for the exclusive use of different domains. We demonstrate how this has a significant effect on the miss rates of these domains and how granting a domain its own share of the TLB isolates it from the TLB demands of other domains. We also suggest the need for an autonomic QoS policy which can utilize the differing TLB requirements of the different domains to maintain the service level QoS guarantees by altering the TLB partition ratios as and when needed. Currently, we are in the process of designing such an autonomic QoS policy.

While it is clear that CR3 tagging will improve the TLB hit ratio, and thereby the performance of VMs, we need a timing model to estimate this benefit in terms of increased throughput and explore how the reduction of flushes will increase the instructions executed per cycle. We are currently developing a timing model to answer these questions.

## 8. ACKNOWLEDGMENTS

This work is sponsored in part by the National Science Foundation under CRI collaborative awards 0751112, 0750847, 0750851, 0750852, 0750860, 0750868, 0750884, and 0751091 and by a grant from Intel Corporation.

## 9. REFERENCES

- [1] *MIPS R400 Microprocessor User's Manual*.
- [2] *The SPARC Architecture Manual Version 9*. Prentice Hall PTR, November 1993.
- [3] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Weigert. Intel virtualization technology for Directed I/O. *Intel Technology Journal*, 10(3):179–192, 2006.
- [4] Advanced Micro Devices. *AMD Secure Virtual Machine Architecture Reference Manual*, December 2008.
- [5] Amd-V nested paging. White paper, AMD, July 2008.
- [6] S. Biemeuller. Asid management in xen amd-V. Xen Summit Spring 2007. Presentation in Xen Summit 2007.
- [7] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.
- [8] U. Drepper. The cost of virtualization. *Queue*, 6(1):28–35, 2008.
- [9] R. Figueiredo, P. Dinda, and J. Fortes. Guest editors' introduction: Resource virtualization renaissance. *Computer*, 38(5):28–31, May 2005.
- [10] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22, New York, NY, USA, 2006. ACM.
- [11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*.
- [12] R. Iyer, R. Illikkal, L. Zhao, S. Makineni, D. Newell, J. Moses, and P. Apparao. Datacenter-on-Chip Architectures: Tera-scale opportunities and challenges in Intel's manufacturing environment. *Intel Technology Journal*, 11(3):227–237, 2007.
- [13] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.
- [14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.
- [15] D. R. Llanos. Tpc-c-uva: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35(4):6–15, 2006.
- [16] M. R. Marty and M. D. Hill. Virtual hierarchies to support server consolidation. *SIGARCH Comput. Archit. News*, 35(2):46–56, May 2007.
- [17] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–178, August 2006.
- [18] K. Nesbit, M. Moreto, F. Cazorla, A. Ramirez, M. Valero, and J. Smith. Multicore resource management. *Micro, IEEE*, 28(3):6–16, May-June 2008.
- [19] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.
- [20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.
- [21] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 2–12, New York, NY, USA, 2006. ACM.
- [22] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., 2005.
- [23] O. Tickoo, H. Kannan, V. Chadha, and R. Illikkal. qtlb: Looking inside the look-aside buffer. In *The 14th International Conference on High Performance Computing*, December 2007.
- [24] TPC Benchmark C Standard Specification Revision 5.10.
- [25] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.