Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted vCPUs

Jiannan Ouyang
University of Pittsburgh
ouyang@cs.pitt.edu

John R. Lange
University of Pittsburgh
jacklange@cs.pitt.edu

Haoqiang Zheng
VMware, Inc
hzheng@vmware.com

Abstract

Virtual Machine based approaches to workload consolidation, as seen in IaaS cloud as well as datacenter platforms, have long had to contend with performance degradation caused by synchronization primitives inside the guest environments. These primitives can be affected by virtual CPU preemptions by the host scheduler that can introduce delays that are orders of magnitude longer than those primitives were designed for. While a significant amount of work has focused on the behavior of spinlock primitives as a source of these performance issues, spinlocks do not represent the entirety of synchronization mechanisms that are susceptible to scheduling issues when running in a virtualized environment. In this paper we address the virtualized performance issues introduced by TLB shootdown operations. Our profiling study, based on the PARSEC benchmark suite, has shown that up to 64% of a VM's CPU time can be spent on TLB shootdown operations under certain workloads. In order to address this problem, we present a paravirtual TLB shootdown scheme named Shoot4U. Shoot4U completely eliminates TLB shootdown preemptions by invalidating guest TLB entries from the VMM and allowing guest TLB shootdown operations to complete without waiting for remote virtual CPUs to be scheduled. Our performance evaluation using the PARSEC benchmark suite demonstrates that Shoot4U can reduce benchmark runtime by up to 85% compared an unmodified Linux kernel, and up to 44% over a state-of-the-art paravirtual TLB shootdown scheme.

Categories and Subject Descriptors D.4.1 [*Process Management*]: Synchronization

Keywords TLB Shootdown, Virtualization, Preemption

VEE '16, April 02-03, 2016, Atlanta, GA, USA.

http://dx.doi.org/10.1145/2892242.2892245

1. Introduction

Several studies have established that the average server utilization in most datacenters is low, ranging from 10% to 50% [2, 9, 12, 14, 18]. While a promising way to improve efficiency is to co-locate multiple virtual machines on the same node in the cloud, the performance overhead introduced by over-commitment inhibits efficient workload co-location. A large body of work has documented the detrimental effects virtual CPU preemption can have on multicore virtual machine performance [10, 11, 13, 15, 16, 20–23]. The majority of this work has focused on the impact of spinlock behaviors, due to the direct effects spinlock delays can have on performance critical code paths. However, relatively little attention has been paid to other sources of local delays caused by preemptions of remote CPU cores. In this paper we focus on the issue of performance overhead caused by TLB operations in the presence of preempted virtual CPU cores (vCPUs).

Cross core TLB operations act as a low level synchronization point in modern Operating Systems in order to maintain consistent application memory mappings. The majority of these operations consist of various cache flushing methods that must be invoked on every CPU in the system. For each TLB flush operation the invoking CPU must wait until the operation has been completed on all other cores before continuing, typically by polling a memory region with kernel preemption disabled. This invocation is achieved by issuing Interprocessor Interrupts (IPIs) to each target CPU, the handlers of which directly invoke a local flush operation. In native environments, these operations have very low latency since at most they only need to wait for a target CPU to exit an atomic region before the IPI is handled. In virtual environments these assumptions no longer hold due to the potential for a target vCPU to be preempted by the underlying host scheduler. This can result in the latencies of TLB flush operations increasing by orders of magnitude depending on the scheduling state of the target vCPUs. We refer to this issue as the *TLB shootdown preemption* problem.

To address this problem we propose Shoot4U, a virtual TLB management mechanism for paravirtualized multicore VMs. Shoot4U eliminates the dependencies on vCPU scheduling states for TLB flush operations and is therefore

able to ensure that TLB operations exhibit consistently low latencies. Shoot4U accomplishes this by intercepting cross vCPU TLB flush operations at the VMM layer, and performing the invalidations directly in the VMM instead of requiring that they be handled inside a guest environment. This optimization allows Shoot4U to avoid any delays caused by a preempted vCPU, and to ensure consistent performance of TLB operations. The Shoot4U mechanism provides a better match for the TLB operation semantics, since at the lowest level it shares the same IPI based signalling behavior as the native versions. This not only allows lower latencies in general, but also eliminates preemption based delays that cause a dramatic increase in the latency variance.

In this paper we make the following contributions:

- An analysis of the impact that various low level synchronization operations have on system benchmark performance.
- Shoot4U: A novel virtualized TLB architecture that ensures consistently low latencies for synchronized TLB operations.
- An analysis of the performance benefits achieved by Shoot4U over current state-of-art software and hardware assisted approaches.

2. Related Work

Most previous work that has looked at the problems associated with VM synchronization overheads has focused on spinlocks and the *lock holder preemption* problem, originally identified by V. Uhlig et al. in 2004 [21]. In that work, the authors proposed a paravirtualization based approach in which the guest OS provides scheduling hints to the underlying VMM. These hints demarcated non-preemptable regions of guest execution that corresponded to critical sections in which a spinlock was held. T. Friebel and S. Biemueller [11] proposed a paravirtual spinlock approach, which was later adopted by Xen and KVM [17]. In their scheme a vCPU notifies the VMM via a hypercall if it has been waiting longer than a threshold. The VMM then blocks the spinning vCPU until the requested lock is released. J. Ouyang and J. Lange [16] identified the *lock waiter preemption* problem existing in queue-based FIFO spinlocks and proposed the preemptable ticket spinlock algorithm (pmtlock) to solve this problem. While spinlock optimizations are an important feature to reduce preemption based delays, they do not address other sources of preemption based delays such as TLB operations, which are the topic of this paper.

There are a few examples of previous work that has looked into the *TLB shootdown preemption* problem. In particular, H. Kim et al. [13] studied the performance degradation caused by both spinlock and TLB shootdown preemptions. They proposed the use of TLB shootdown IPIs as a VMM scheduling heuristic in order to reduce the delay introduced by a preempted vCPU. While their approach does help alleviate the delays imposed by TLB shootdowns on preempted vCPUs, it does not address the underlying problem directly. In contrast, our Shoot4U mechanism addresses the source of the problem directly by eliminating the necessity for busy-waiting inside the VM.

Most relevant to our work is the KVM paravirtual remote flush TLB scheme (kvmtlb) developed by the Linux community [4]. This scheme maintains the preemption state of all vCPUs inside the VMM and shares this information with the guest. When initiating TLB operations, if the remote vCPU is running, then the conventional shootdown approach is used. Otherwise, if the remote vCPU is preempted, a *should_flush* flag is set on that remote vCPU and an IPI is not sent. When rescheduling a vCPU, the VMM checks the *should_flush* flag. If set, the VMM invalidates all TLB entries of that vCPU. While this approach does address the underlying problem, it still possesses a number of shortcomings. First it still imposes the overheads of IPI routing between vCPUs which Shoot4U eliminates. Second, in an overcommitted environment it is possible that the preemption state of a vCPU can change after its state has been checked by the invoking CPU but before the IPI is actually delivered. As shown in our evaluation, the worst-case TLB shootdown latency of Shoot4U is an order of magnitude better than kvmtlb.

Other approaches to improving VM performance in the face of cross core synchronizations include improving VMM scheduling policy. H. Kim et al. [13] proposed demand-based coordinated scheduling that controls time-sharing in response to inter-processor interrupts (IPIs) between virtual CPUs. Other work proposed the use of co-scheduling [15]. However, strict co-scheduling is not scalable and may result in CPU fragmentation issues, which has led to more relaxed co-scheduler approaches as seen in VMware ESX [8]. Other co-scheduling variants include adaptive co-scheduling schemes [22, 23] that allow the VMM scheduler to dynamically alternate between co-scheduling and asynchronous scheduling for a particular VM, as well as balanced scheduling [20] which associates a VM's individual vCPUs with dedicated physical CPUs and does not require that the vCPUs be co-scheduled. While each of these approaches alleviate the problems caused by intra-VM synchronizations, they do so by providing workarounds as opposed to addressing the underlying issues.

3. The TLB Shootdown Preemption Problem

Translation Look-aside Buffers (TLBs) are a critical hardware component for virtual memory based systems, however they still require explicit management by the Operating System (OS) in order to maintain cache coherence. This requires the OS itself to directly manage the contents to the TLB caches on each CPU core in the system by ensuring that stale entries are removed before they can be accessed by any hosted applications. This is especially a problem for

multi-threaded applications as they leverage shared page tables both as a space saving optimization as well as a way to amortize address space management overheads. Cache coherence is managed by the OS through the use of invalidation and flushing operations that remove one or more entries from a local TLB cache. The operations are propagated to other cores in the system via IPI based signalling that directly invoke a given TLB operation on a remotely targeted CPU, a mechanism that is canonically referred to as a *TLB shootdown*.

Modern operating systems consider TLB shootdown operations to be performance critical and so optimize them to exhibit very low latency. The implementation of these operations is therefore architected to ensure that shootdowns can be completed with very low latencies through the use of IPI based signalling. As such, TLB shootdowns can be implemented using a busy wait based stall of the invoking CPU while the operation is handled on each of the remote CPUs. Unfortunately, the low latency provided by IPI handlers is only ensured when the target CPU is available to handle the resulting interrupt. While this is generally a reasonable assumption in native environments, it does not carry over to virtualized environments as the availability of a given vCPU to handle interrupts is entirely dependent on the behavior of the underlying host scheduler.

The *TLB shootdown preemption problem* is the result of a vCPU invoking a TLB shootdown on a remote vCPU that is currently preempted by the host scheduler and therefore not available to handle the resulting IPI interrupt. In this case the invoking vCPU will block in a polling based loop until the target vCPU is rescheduled and returns to an active state. The scheduling delays, or the time between the preemption and rescheduling of a vCPU, are often orders of magnitude larger than the latency that TLB shootdown operations were designed for. This is especially true when multiple vCPUs are sharing the same underlying physical CPU. These unexpected delays can cause significant impacts on application performance depending on the workload, with particularly dramatic effects seen on multi-threaded workloads that require large amounts of address space modifications.

3.1 Performance Analysis

In order to better understand the effects that vCPU preemption has on TLB operations as well as other low level operations, we measured the performance degradation caused by CPU overcommitment on the PARSEC [5] benchmark suite. Specifically, we used the Linux perf [6] tool set to measure the percentage of time spent in various kernel functions.

We first measured the PARSEC performance using a single KVM based guest without overcommitting resources (1-VM), before adding a second KVM guest on the same machine running a CPU bounded workload using sysbench [7] (2-VMs). Each VM was configured to use the same number of vCPUs (12) evenly distributed on the underlying physical CPUs, so that each physical core was shared by two vCPU cores from different VMs. Equal time sharing between vCPUs was ensured using Linux cgroups [1] and the Pause-Loop Exiting (PLE) [19] feature was disabled.

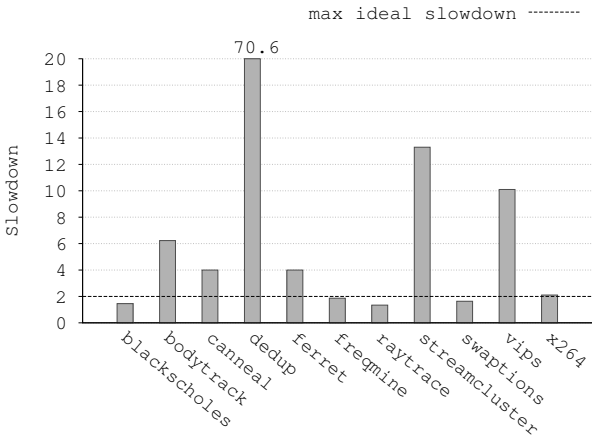


Figure 1: Performance Slowdown of CPU Overcommitment

Figure 1 shows the benchmark results for the 1-VM and 2-VMs configurations. The ideal slowdown would be 2x, due to the equal time sharing configuration of each physical CPU. As the results show 6 out of the 11 benchmarks have performance slowdowns of over 4x; 3 exhibit more than 10x slowdown; and the dedup benchmark has a slowdown of 70.6x.

For each of the applications we separated out the overheads resulting from TLB shootdowns (k:tlb) as well as spinlocks (k:lock), being the two most common causes of preemption based performance problems. The remaining overhead was split between other kernel level functions (k:other) and time spent in userspace (u:*). Figure 2 shows these results. With the exception of dedup, all benchmarks spent the majority of time in userspace for the 1-VM scenario. However, for the 2-VMs case a significant number of benchmarks exhibit noticeable increases in kernel based overheads. For dedup in particular, 64% of the CPU time is spent on TLB shootdown operations. As the results show, overheads resulting from spinlocks and TLB shootdowns account for the majority of the added overhead.

Next, we instrumented the Linux kernel using ktap [3], and measured the latency of TLB shootdown operations directly when running the dedup benchmark. Figure 3 shows the cumulative distribution function (CDF) of TLB shootdown latencies using a logarithmic X axis. As the figure shows, the 2-VMs case exhibits a significant increase in the average operation latency with the 90th percentile increasing by two orders of magnitude over the 1-VM configuration.

4. Shoot4U

To address the TLB shootdown preemption problem we present Shoot4U, a paravirtual TLB management interface. Shoot4U uses a VMM-assist technique to optimize

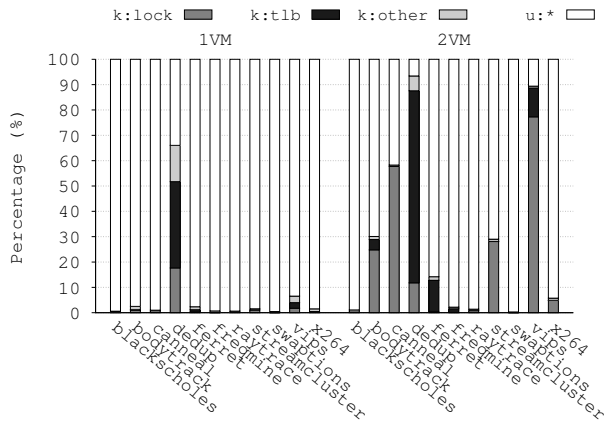


Figure 2: CPU Usage Profiling

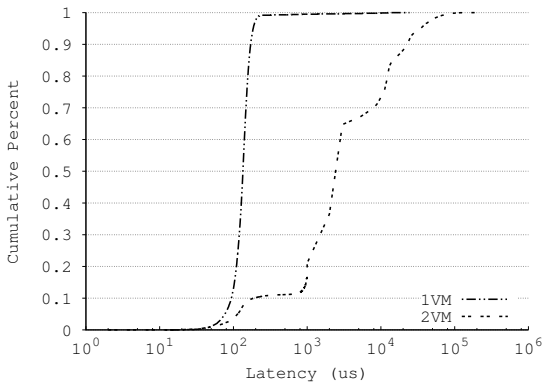


Figure 3: CDF of TLB Shootdown Latency

TLB shootdowns by performing the invalidations inside the VMM itself without the need to invoke or signal the guest OS. Shoot4U relies on hardware instructions available as part of the virtualization extensions on modern x86 based CPU architectures. These instructions allow targeted invalidation of TLB entries that belong to a specific VM environment.

Before explaining how Shoot4U works, it is necessary to understand how a conventional TLB shootdown operation works in a virtual environment. To initiate a TLB shootdown operation, the invoking vCPU sends an IPI with a specific vector number to a set of target vCPUs. The invoker then enters a polling loop until all receiver vCPUs have processed and acknowledged the requests by setting a flag located in shared memory. The transmission of the IPI by the vCPU causes the hardware to trap into the underlying VMM where it can be emulated, ultimately resulting in the VMM generating a new IPI that is actually transmitted to the VMM on the physical CPUs hosting the targeted vCPUs. Typically an IPI delivery by the hardware will indirectly cause any running VM to trap, so that the underlying host system software can handle it. In this case the IPI is handed off to the VMM, which completes the IPI emulation by delivering an IPI to the targeted vCPU via the injection of a virtual interrupt. After the virtual interrupt has been injected, it will be handled as soon as the target vCPU resumes execution.

Shoot4U is based on the observation that modern hardware allows the underlying VMM to perform the invalidation operation internally, thus removing the need to inject a virtual interrupt into the target vCPU. Current x86 processors from both Intel and AMD support the use of Virtual Processor IDs or VPIDs (Intel) and Address Space IDs or ASIDs (AMD) to tag TLB entries with a given ID assigned to a VM context. Our implementation targets the Intel architecture (using KVM/Linux 3.16), but there is nothing preventing the same approach from being used on an AMD based system. Along with the ability to tag TLB entries with an associated VPID/ASID these CPUs support a new set of invalidation instructions (e.g. *invpid*) that selectively flush TLB entries based on a given ID tag. These instructions can be executed by the VMM itself, without any involvement of the VM's guest OS. Therefore, instead of relying on IPI injection as described above, Shoot4U enables the VMM to process the TLB invalidation request immediately by invalidating guest TLB entries itself.

Our implementation of Shoot4U introduces a paravirtual hypercall interface that replaces the existing IPI based TLB shootdown mechanism. In Shoot4U, the invoking vCPU issues a hypercall down to the underlying VMM with the target vCPUs and address range being invalidated specified as parameters. Upon trapping into the hypercall handler, the VMM determines the set of physical CPUs that are currently hosting the set of vCPUs, and issues a physical IPI to each of them. These IPIs are handled by the VMM itself, which then executes the appropriate set of invalidation operations internally without any interaction with the VM context. While the VMM handles the invalidations for the target vCPUs, the VMM on the invoking CPU polls for completion in a busy wait loop. Once the operations complete the VMM then returns from the hypercall and the VM resumes operation. While superficially it might appear that we have just moved the polling loop from the guest into the VMM, it should be noted that operation completion is no longer dependent on host scheduler behaviors since it does not have to wait for a vCPU to be running in order to complete.

Figure 4 shows the paravirtual hypercall interface provided by a KVM host with Shoot4U support. To utilize this interface, the guest VM needs to specify the hypercall ID, a bitmap of targeted vCPUs, and the address range being invalidated. Our current implementation of Shoot4U supports up to 64 vCPUs due to the size of the bitmap. However, we can easily support more vCPUs by mapping the bitmap into memory.

5. Evaluation

We evaluated Shoot4U on a dual socket Dell R450 server configured with Intel "Ivy-Bridge" Xeon processors (6 cores each) with hyperthreading enabled and 24 GB of RAM split across two NUMA domains. Each server was running CentOS 7 with Linux Kernel 3.16 with a modified version of KVM implementing the Shoot4U interface. We performed the evaluation using 2 separate VMs each with 12 vCPUs both mapped to the same socket. Each vCPU was pinned to a single hyperthreaded CPU core, so that each core was shared by 2 vCPUs. The Linux cgroups [1] interface was used to allocate an equal share of CPU time to each VM. One VM (VM1) was configured to run the PARSEC benchmark suite [5], while the other (VM2) ran a CPU bounded competing workload based on sysbench [7]. We used the Linux default TLB shootdown scheme with as the baseline, and compared it with Shoot4U as well as the current TLB shootdown optimization provided by KVM, denoted kvmtlb [4].

5.1 TLB Shootdown Latency

		baseline	kvmtlb	shoot4u
1VM	Mean	166	122	28
	Max	24,428	9,953	453
2VM	Mean	9,048	5,401	22
	Max	194,108	126,923	15,034

Table 1: TLB Shootdown Latency (usec)

The first experiment used ktap [3] to measure the completion time of TLB shootdown requests in the guest, running the dedup benchmark from PARSEC. The results are shown in Table 1, including the average and maximum completion time both with and without a 2nd VM sharing a physical CPU. It shows that both kvmtlb and Shoot4U significantly improve TLB shootdown performance in both cases. However, Shoot4U outperforms the other schemes: it is 4.3 and 245.5 times faster than kvmtlb on average for the 1-VM and 2-VMs cases respectively. Its worse case performance is also order of magnitude better than others.

Figure 5 shows the cumulative distribution function (CDF) of TLB shootdown latencies from the same experiment. Shoot4U not only provides better overall performance, but also exhibits much less variance than other approaches. In a non-overcommitted configuration Shoot4U provides superior performance by reducing the overheads of a TLB shootdown by reducing the number of world switches

```
kvm_hypercall3(unsigned long KVM_HC_SHOOT4U,
  unsigned long vcpu_bitmap,
  unsigned long start,
  unsigned long end);
```

Figure 4: Shoot4U API

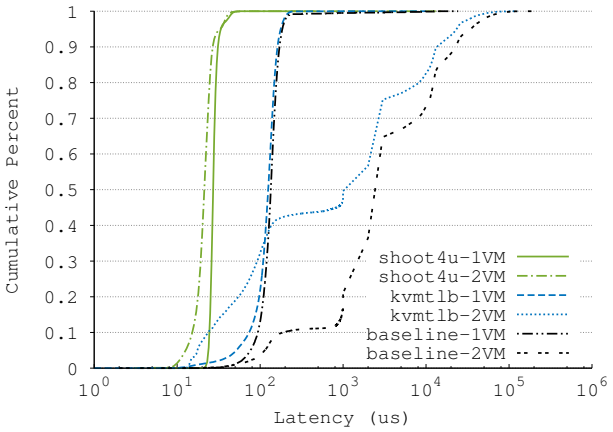


Figure 5: CDF of TLB Shootdown Latencies

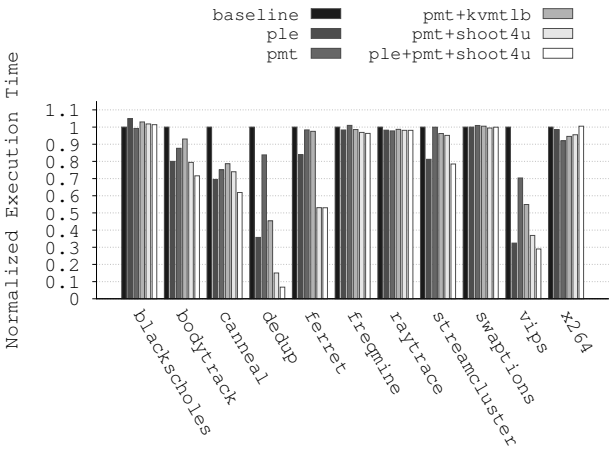
needed for IPI propagation. Moreover, Shoot4U is able to maintain consistent performance in an overcommitted configuration, while the other solutions experience slowdowns due to vCPU preemptions, which Shoot4U is immune to.

5.2 Macro-Benchmark Performance

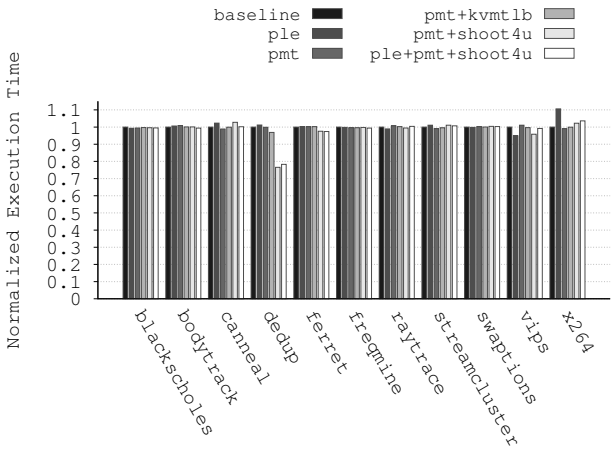
Our next experiment evaluated various TLB shootdown schemes using multi-threaded benchmarks from the PARSEC [5] benchmark suite. Each configuration was evaluated 3 times, and the average is reported. We also incorporated an optimized spinlock mechanism based on the preemptable ticket spinlock (PMT) algorithm [16]. This allowed us to compare the performance impact of spinlock based locking versus TLB operations. We also studied the impact of Pause-Loop Exiting (PLE), a hardware assisted spinning detection and optimization feature supported by KVM and recent Intel processors.

Figure 6 shows the normalized execution time of each benchmark using a sweep of various configurations. In the 2-VMs case on the left, it can be observed that Shoot4U achieves the best performance on almost all benchmarks. It outperforms kvmtlb by more than 10% on 4 benchmarks, and in the best cases, it is 85% faster than the baseline on dedup, and 44% faster than kvmtlb on ferret. It can also be observed that PLE yields pretty good performance improvements on many benchmarks; moreover, further improvements can be achieved when combined with Shoot4U. In the 1-VM case, performance of various schemes are comparable as the preemption rate is low when the system is not over-committed. However, Shoot4U still achieves about 20% performance improvement on dedup, which is the most TLB shootdown intensive workload. It is also notable that enabling PLE introduces about 10% overhead on x264 in this case.

Finally we reran the experiments from Section 3.1 in order to compare the reduction of synchronization overheads possible using Shoot4U and optimized spinlocks based on

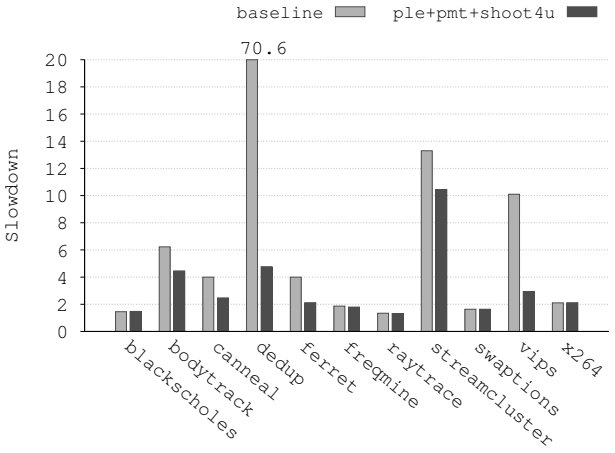


(a) With Competing VM (2-VMs)

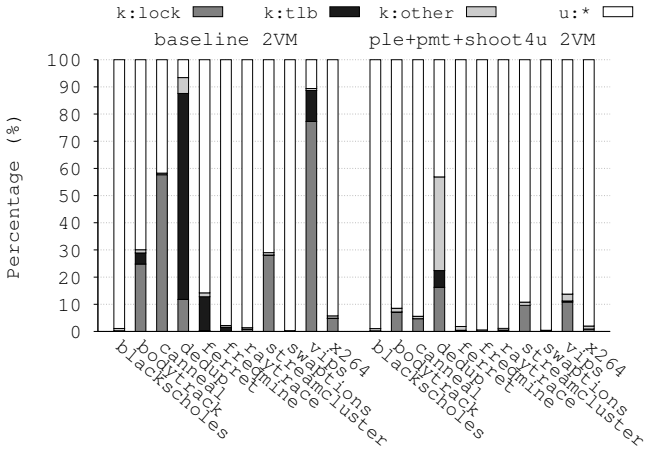


(b) Without Competing VM (1-VM)

Figure 6: PARSEC Performance Evaluation with Shoot4U



(a) Performance Slowdown of CPU Overcommitment



(b) CPU Usage Profiling

Figure 7: PARSEC Performance Analysis

PMT. Figure 7 (a) compares the slowdown of both the baseline and optimized configurations in the 2-VMs scenario. Significant performance improvement is observed on 6 out of the 11 benchmarks. For dedup and vips in particular, the slowdown decreases from 70.6 to 4.8 and from 10.1 to 2.9 respectively.

Figure 7 (b) provides a profile of the sources of overheads for the two configurations. There are significant reductions of kernel based overhead for all kernel intensive benchmarks, explaining the overall performance improvements for those benchmarks. Furthermore, for nearly every benchmark the time spent in TLB related functions is almost eliminated, with the exception of dedup which is still greatly reduced.

6. Conclusion

This paper presents Shoot4U, an approach to optimizing TLB operations across virtual CPUs allocated to a given virtual machine. We conducted a set of experiments in order to provide a breakdown of overheads caused by preempted virtual CPU cores, showing that TLB operations can have a significant impact on performance with certain workloads. To address that problem we introduced Shoot4U, an optimization for TLB shootdown operations that internalizes the operation in the VMM and so no longer requires the involvement of a guest's vCPUs. Our evaluation of Shoot4U demonstrates the effectiveness of our approach, and illustrates how under certain workloads our approach is dramatically better than current state of the art techniques.

References

[1] Linux Control Groups (cgroups). https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[2] Gartner Says Efficient Data Center Design Can Lead to 300 Percent Capacity Growth in 60 Percent Less Space. http://www.gartner.com/newsroom/id/1472714.

[3] ktap: A lightweight script-based dynamic tracing tool for Linux. http://www.ktap.org/.

[4] KVM Paravirt Remote Flush TLB. https://lwn.net/Articles/500188/.

[5] The PARSEC Benchmark Suite. http://parsec.cs.princeton.edu/.

[6] perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org/.

[7] Sysbench. https://github.com/akopytov/sysbench.

[8] Vmware(r) vsphere(tm): The cpu scheduler in vmware esx(r) 4.1. Technical report, VMware, Inc, 2010.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2013.

[10] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *Proc. 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, Philadelphia, PA, June 2014. USENIX Association. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ding.

[11] T. Friebel. How to Deal with Lock-Holder Preemption. Presented at the Xen Summit North America, 2008.

[12] J. Kaplan, W. Forrest, and N. Kindler. Revolutionizing Data Center Energy Efficiency. Technical report, McKinsey & Company, 2008.

[13] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based Coordinated Scheduling for SMP VMs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proc. of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15, 2015. . URL http://doi.acm.org/10.1145/2749469.2749475.

[15] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd International Conference on Distributed Computing Systems*, 1982.

[16] J. Ouyang and J. R. Lange. Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proc. 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2013.

[17] K. Raghavendra and J. Fitzhardinge. Paravirtualized ticket spinlocks, May 2012. URL http://lwn.net/Articles/495597/.

[18] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proc. 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012. ISBN 978-1-4503-1761-0. . URL http://doi.acm.org/10.1145/2391229.2391236.

[19] R. v. Riel. Directed yield for pause loop exiting, 2011. URL http://lwn.net/Articles/424960/.

[20] O. Sukwong and H. S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proc. 6th European Conference on Computer Systems (EuroSys)*, 2011.

[21] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proc. 3rd conference on Virtual Machine Research And Technology Symposium*, 2004.

[22] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic Adaptive Scheduling for Virtual Machines. In *Proc. 20th International Symposium on High Performance Parallel and Distributed Computing (HPDC)*, 2011.

[23] L. Zhang, Y. Chen, Y. Dong, and C. Liu. Lock-Visor: An Efficient Transitory Co-scheduling for MP Guest. In *Proc. 41st International Conference on Parallel Processing (ICPP)*, 2012.