# Overcoming Virtualization Overheads for Large-vCPU Virtual Machines

Ozgur Kilic, Spoorti Doddamani, Aprameya Bhat, Hardik Bagdi, Kartik Gopalan
Contact: {okilic1,sdoddam1,abhat3,hbagdi1,kartik}@binghamton.edu

***Abstract*—Virtual Machines (VM) frequently run parallel applications in cloud environments, and high performance computing platforms. It is well known that configuring a VM with too many virtual processors (vCPUs) worsens application performance due to scheduling cross-talk between the hypervisor and the guest OS. Specifically, when the number of vCPUs assigned to a VM exceeds available physical CPUs then parallel applications in the VM experience worse performance, even when number of application threads remains fixed. In this paper, we first track the root cause of this performance loss to inefficient hypervisor-level emulation of inter-vCPU synchronization events. We then present three techniques to minimize hypervisor-induced overheads on parallel workloads in large-VCPU VMs. The first technique pins application threads to dedicated vCPUs to eliminate inter-vCPU thread migrations, reducing the overhead of emulating inter-processor interrupts (IPIs). The second technique para-virtualizes inter-vCPU TLB flush operations. The third technique enables faster reactivation of idle vCPUs by prioritizing the delivery of rescheduling IPIs. Unlike existing solutions which rely on heavyweight and slow vCPU hotplug mechanisms, our techniques are lightweight and provide more flexibility in migrating large-vCPU VMs. Using several parallel benchmarks, we demonstrate the effectiveness of our prototype implementation in the Linux KVM/QEMU virtualization platform. Specifically, we demonstrate that with our techniques, parallel applications can maintain their performance even when 255 VCPUs are assigned to a VM running on only 6 physical cores.**

***Index Terms*—Virtualization, Virtual Machines, Virtual CPUs**

## I. Introduction

Hypervisors [1], [2] virtualize the hardware resources for virtual machines (VMs), including number of virtual processors (vCPUs), amount of memory, storage, and network bandwidth. Currently, elasticity of VMs is managed by an orchestration engine, which decides how much hardware to allocate to a VM, on which physical machine a VM runs, and when it is migrated. These allocations are currently managed at coarse granularity when creating or migrating VMs.

A VM could theoretically provide "unlimited" resources to its applications and the underlying hardware resources can be transparently marshaled at finer-granularity to meet the application's needs. A requirement for such an elastic VM is that the abstraction of unlimited resources should not negatively impact application performance. In our findings, this is not the case with current virtualization techniques which scale poorly as the amount of virtualized resources assigned to a VM increase.

It is widely known that virtualization introduces overheads that negatively impacts application performance [3], [4], [5], [6]. Although architectural support for virtualization [7], [8], [9] helps to some extent, it does not eliminate all virtualization overheads. For processor virtualization, which is our focus, the hypervisor introduces overheads in emulating inter-processor interrupts (IPIs) [10], TLB invalidation, and transitions to/from idle modes, among others. Semantic gap between the guest OS and hypervisor also causes *double scheduling*, where the guest and host-level CPU schedulers make independent, but inefficient, scheduling decisions. Many prior efforts [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], have investigated these overheads. However, the impact of these overheads on VMs having large number of vCPUs has not been systematically addressed.

VM orchestration is another aspect of cloud platforms to perform flexible placement and movement of VMs across machines in a data center. A VM may be migrated to a machine with fewer resources for consolidation when the VM does not fully utilize all assigned resources. Conversely, to handle an increase in workload, a VM may be migrated to a machine with more resources. Currently, cloud providers and users need to guess the ideal number of vCPUs, memory capacity, and I/O capacity with which to configure a VM relative to the corresponding hardware resources in the physical machine. Focusing on processors, when a VM migrates to another machine with different number of physical processors (pCPUs), the cloud provider may need to either reconfigure and reboot the migrated VM with a new vCPU count, or use heavyweight CPU hotplug/unplug [22] mechanisms. Despite improvements [23], [24], both rebooting and hotplugging lead to significant overheads and disruption for applications.

This paper focuses on the performance of parallel workloads in a VM having large number of vCPUs. *We observe that when more vCPUs are assigned to a VM than the available physical CPUs (pCPUs), the performance of parallel applications running in the VM degrades, even for a constant workload.* We trace the root causes of this performance loss to the emulation of inter-vCPU synchronization events and CPU scheduling mismatch between the hypervisor and the guest. We present a solution that decouples a parallel application's performance in a VM from the number of vCPUs assigned to the VM and provides a more predictable application performance.

The key idea is to configure a VM with a large number of vCPUs, ideally one dedicated vCPU per application thread[1].

[1]In practice, we configure maximum vCPUs allowed by the underlying hypervisor implementation, which is currently 255 vCPUs in KVM/QEMU [25].

Increasing the number of vCPUs also increases the overhead of emulating inter-vCPU synchronization events in the hypervisor, specifically Inter-processor Interrupts (IPIs) between vCPUs. We develop three techniques to reduce hypervisor-level scheduling and IPI emulation overheads for large-vCPU VMs.

1) First, we obviate the need for most vCPU scheduling in the guest OS by assigning each application thread to a dedicated vCPU. Dedicating a vCPU to each thread eliminates the IPIs needed to synchronize inter-vCPU thread migration, and the corresponding emulation and mode switching (VM Exit/Entry) costs.
2) Second, we implement a paravirtualized TLB flushing mechanism for vCPUs that is less expensive than intercepting and emulating IPIs for inter-VCPU TLB flush operations.
3) Finally, we speed up the mechanism for reactivating idle vCPU threads from blocked state in the hypervisor by prioritizing the delivery of rescheduling IPIs to idle vCPUs.

These techniques together enable a VM configured with maximum allowable vCPUs to maintain the best application performance possible on available pCPUs. The orchestration engine also regains the flexibility to migrate VMs to a machine that has more or fewer pCPUs. More pCPUs in the new machine translates to better performance, since a parallel application can now exploit increased parallelism by virtue of the VM having more vCPUs that can execute in parallel. Conversely, if the new machine has fewer pCPUs, our approach enables the VM to scale down its application performance without incurring the hypervisor-level emulation costs discussed earlier. In contrast, earlier approaches to address double scheduling [19], [20] are overly conservative since a VM cannot have more vCPUs than pCPUs in the machine. This prevents a large-vCPU VM from being migrated to a machine with fewer pCPUs, or a small-vCPU VM from exploiting increased parallelism when migrated to a machine with more pCPUs.

The scope of this paper is focused on a single large-VCPU VM's performance and we defer the consideration of multiple co-resident VMs to future work. The rest of the paper is organized as follows. Section II discusses IPI and double scheduling overheads in more detail. Sections III and IV present the design and implementation of our techniques. Section V presents the evaluation of our prototype using Parsec benchmarks. Section VI discusses related work and Section VII concludes the paper.

## II. Processor Virtualization Overheads

In this section we further discuss overheads in processor virtualization due to IPIs and double scheduling.

Inter-processor interrupts (IPIs) [10] are used by processor hardware to communicate with each other. When a CPU wants another CPU to perform certain action, it interrupts the current running task on that CPU by sending an IPI to preempt the current task from a CPU and assign a specific target task. The CPU which receives the IPI immediately pauses its job and handles the IPI. IPI processing between pCPUs in hardware does not take significant time. However, for a VM, IPI between vCPUs must be emulated in software in the hypervisor, which is slower. When an IPI is sent to a vCPU in a VM, it triggers a VM Exit, and the interrupt is then forwarded to the destination vCPU by the hypervisor. Suppose, the receiver vCPU is not scheduled on a pCPU, then the hypervisor needs to first schedule it on a pCPU in order to inject the pending IPI which gets processed in guest mode. Some IPIs, such as *function call IPIs*, require a response from each receiver vCPU, which increases the IPI processing overhead due to delay in waiting for all responses. A rescheduling IPI forces the receiver to invoke the CPU scheduler to schedule a new task. This causes significant overhead on the system as the scheduling of a task may get delayed until the IPI is delivered to destination vCPU and processed. There are several studies in the literature to optimize IPI delivery [18], [26], [27] but none of them addresses overheads arising from large number of vCPUs and parallel workloads.

Another overhead that arises with virtualization is double scheduling. Guest OS that runs in the VM is unaware of the virtualization environment provided by the hypervisor layer, which runs in the host OS. When guest OS schedules processes according to its scheduling policy, it tries to distribute the entire workload of all the processes among all of its vCPUs. The host OS performs another level of scheduling among host-level processes and threads. The hypervisor in the host OS creates and manages vCPUs for guests. At the host-level, a vCPU is essentially a thread that executes in the guest mode and is scheduled on a pCPU just like any other host-level processes. A vCPU thread transitions between execution states running, ready or blocked. The CPU scheduler in the host OS makes scheduling decisions considering all vCPU threads from all the VMs that are being hosted. A vCPU in guest mode may be idle, or executing a guest-level thread, or executing code in the guest kernel. Due to semantic gap [28] between guest and hypervisor, the host-level scheduler is unaware of the guest scheduler. A scheduling decision in the guest OS can be overridden by another scheduling decision in the host OS, leading to sub-optimal performance in the guest. For instance, a guest vCPU may steal work, i.e. migrate a ready thread, from another vCPU for better CPU utilization. However, the hypervisor may preempt the work stealing vCPU to schedule another vCPU, defeating the purpose of thread migration. Hypervisor may also preempt a running vCPU which is holding a spin lock [29] causing a problem called lock holder preemption [11], in which other vCPUs contending the spin lock are delayed.

Both inter-VCPU synchronization and double scheduling overheads grow as the number of vCPUs assigned to a VM is increased. In the next section, we present our solutions to address these overheads followed by their evaluation.

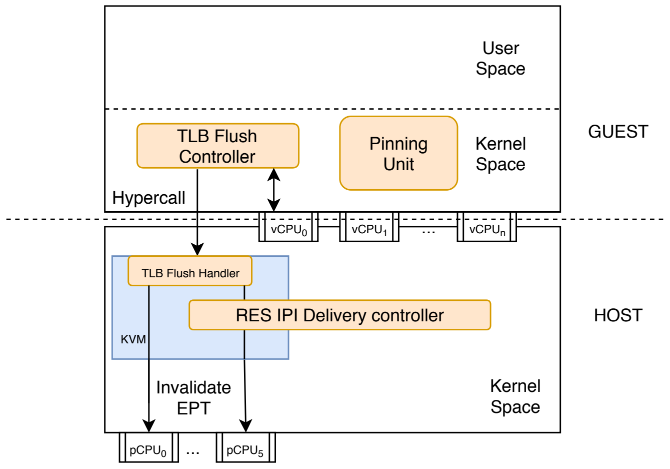

Fig. 1. Overview of techniques used to reduce hypervisor-level emulation overheads for large vCPU virtual machine.

## III. Design

Support for large-vCPU VMs presents two major challenges. First, hypervisor-induced overheads increase as the number of vCPUs increase. Hence the first design requirement is to maintain application performance when the number of vCPUs exceeds the number of pCPUs. The second challenge is to support unmodified applications while minimizing any changes to the guest OS. Figure 1 illustrates our design, which has three main components: (a) assigning a dedicated vCPU to every application thread to eliminate inter-vCPU thread migration, (b) para-virtualizing TLB-flush operations by the guest to reduce traditional emulation overhead, and (c) speeding up the reactivation of idle vCPUs for newly woken threads by prioritizing the delivery of rescheduling IPIs. These techniques are described in detail below.

### A. Allocating a Dedicated vCPU per Thread

Traditional operating systems allow users to create (practically) unlimited number of threads. The native CPU scheduler is responsible for dynamically multiplexing thread execution on pCPUs at runtime. When applications run in a VM, the host OS scheduler no longer has the ability to directly schedule application threads. Instead, it schedules a small number of vCPUs belonging to the VM, and the guest OS schedules threads on vCPUs. While this arrangement allows division of labor between the guest and the host, it also brings virtualization overheads due to double scheduling, as discussed earlier.

Imagine if, instead of the guest OS scheduling threads to intermediate vCPUs, the host OS could directly schedule guest threads on pCPUs. This would eliminate double-scheduling problems discussed earlier.

To eliminate two levels of vCPU and pCPU scheduling, we propose to dedicate one vCPU per guest thread[2]. Thus the guest OS would have no need to perform any CPU scheduling, since now the thread-to-vCPU mapping is fixed for the lifetime of the thread. On the other hand, the host OS sees each guest thread as a distinct schedulable entity in the form of the corresponding vCPUs. Unfortunately, a large number of vCPUs required for this solution end up increasing hypervisor-induced overheads, as discussed before. In the following two subsections, we discuss ways to limit this overhead.

[2]Currently KVM supports 255 maximum number of vCPUs for a guest [25].

### B. Para-virtualizing Inter-vCPU TLB Flush Requests

TLB flush operation is used by the OS to clear any cached page table mappings in the TLB of one or more CPU cores. Generally, any changes to virtual address mapping of a process may invoke TLB flush to ensure that stale mappings are evicted from all CPU cores. One CPU core may request other cores to flush their TLBs by sending a synchronous *function call* IPI that invokes the TLB flush operation on the remote cores; we call this IPI as TLB flush IPI for simplicity. On a native OS, the IPI delivery is handled completely in the CPU execution hardware.

However, for a VM, the TLB flush IPI from one vCPU to other vCPUs must be emulated by the hypervisor which alone knows the vCPU to pCPU mapping that is needed for IPI delivery. TLB flush IPI also requires the IPI sender to wait until all receivers acknowledge the flush operation. If one of the IPI receiving vCPUs is delayed in being scheduled by the host OS, the sender vCPU would have to wait longer until the TLB flush IPI is acknowledged.

Our second technique attempts to reduce the overhead of emulating the delivery of TLB flush IPI between vCPUs. Instead of delivering a TLB flush IPI to every vCPU, we use KVM's hypercall mechanism – an explicit service request from the guest OS to the hypervisor – to perform TLB flush operation directly on all pCPUs. For a large-vCPU VM, this improvement eliminates the overhead of having to inject a virtual IPI into each of the 255 vCPUs.

### C. Accelerating Idle vCPU Reactivation

When a pCPU has no ready threads in its scheduling queue, it enters idle mode which places the CPU in low-power state. Rescheduling IPIs are used in Linux to wake up a CPU from idle mode to schedule a new ready thread.

In a VM, an idle vCPU cannot place the pCPU in low-power mode because other vCPUs may need to execute on the pCPU. Hence an idle vCPU exits guest mode (via VM Exit) back to the hypervisor, which emulates idling by moving the corresponding vCPU thread to blocked state. When a guest thread becomes ready to execute on the idle vCPU, a rescheduling IPI is generated from a peer vCPU. The time to deliver this rescheduling IPI to the blocked vCPU and re-enter guest mode increases when there are more vCPUs. The current Linux CPU scheduler in the host OS does not prioritize rescheduling the reactivated idle vCPU and may schedule other vCPUs instead.

The hypervisor normally moves the blocked (idle) vCPU thread to ready state and triggers the host-level Linux CPU scheduler which executes its own default scheduling algorithm.

This normally works well when the pCPU runs only one vCPU but not when pCPUs are over-subscribed. Since the Linux CPU scheduler in host OS and the KVM hypervisor do not coordinate, it delays the reactivation of a newly woken vCPU. This scheduling delay defeats the main purpose of generating a rescheduling IPI for the idle vCPU.

To reduce the latency of reactivating the idle vCPU, we modify the hypervisor-level mechanism to block and wake up an idle vCPU. Specifically, when moving the idle vCPU from blocked to ready state, we temporarily boost the scheduling priority of the idle vCPU thread so that the CPU scheduler can schedule immediately and the rescheduling IPI can be delivered in a timely fashion.

## IV. Implementation

Our implementation uses KVM/QEMU virtualization platform in the Linux operating system with Linux kernel version 4.12.9 and QEMU version 2.5.0. KVM is the hypervisor that executes as a kernel module in the Linux kernel and QEMU is a per-VM management process that runs in user-space of the host OS. QEMU and KVM coordinate with each other to carry out the initialization and runtime management of the VM's vCPU, memory, and virtual I/O devices.

Our implementation has three major components; a guest kernel mechanism to pin each new application thread to a dedicated vCPU, a para-virtualized TLB flush mechanism, and a KVM hypervisor to accelerate the delivery of rescheduling IPI to idle VMs. Below we describe the implementation details of each of these components.

### A. Dedicated Per-Thread vCPUs

The main idea is to pin each thread to a dedicated vCPU and prevent the guest CPU scheduler from performing work stealing, which means to migrate threads across vCPUs during runtime to balance loads. With large-VCPU VMs, pinning obviates the need for work stealing, since only one active application thread runs on each vCPU, hence all vCPUs are balanced by design. Absence of work stealing in the guest implies that no rescheduling IPIs need be generated for inter-vCPU load balancing. We also disable a kernel-level load balancing mechanism [30], called `sched_load_balance` to prevent Linux scheduler from migrating threads to equalize loads across vCPUs.

That said, in Linux, some OS services create one kernel-level thread per CPU. Hence the pinned application thread may share its vCPU with threads meant for system level services. In our experience these kernel threads run only for occasional tasks and do not contend much with the pinned application threads. Hence the guest CPU scheduler runs only when a new thread is started and when switching between kernel services and the application thread.

An application thread is pinned to its own vCPU at thread creation time (in the `fork` system call). Ideally, we want only one application thread per vCPU. Practically, the KVM hypervisor supports a maximum of 255 vCPUs, due to an 8-bit APIC ID field in the local APIC. Hence, if there are more than 255 application threads at any instant, then some threads may be pinned to the same vCPU. A more recent hardware extension for APIC, called x2APIC [31], [26], supports a 32-bit process ID field and can enable fully dedicating one vCPU per application thread.

### B. Para-virtualizing TLB Flush

Operating system uses TLB [2] to speed up memory translations. Whenever the memory mappings are updated in the TLB of a CPU, corresponding mappings in the other CPUs need to be invalidated to maintain consistency in translations. To flush the TLB mappings on other CPUs, the CPU modifying any page table mappings notifies all affected CPUs by sending a function-call IPI which triggers each receiver CPU to execute a function that invalidates the affected mappings in the local TLB. We call this IPI as *TLB flush IPI*. In Linux, the CPU sending the TLB flush IPI must wait till it receives acknowledgments from all receiver CPUs (via an in-memory data structure) that they have invalidated the affected TLB mappings. On bare metal, TLB flush IPIs are delivered within the execution hardware and do not incur any noticeable delays in the software.

However for a VM, the hypervisor cannot ensure that all sender and receiver vCPUs are simultaneously scheduled on pCPUs. Hence the KVM hypervisor needs to emulate IPI delivery to each receiver vCPU that is not already running at the time IPI is generated (meaning that the receiver vCPU was either blocked or pre-empted). Such receiver vCPUs are first woken up, if blocked for idling, and scheduled on a pCPU by the host-level CPU scheduler. Then the hypervisor injects the IPI notification in a vCPU-specific data structure and transitions the vCPU to guest state (via VM Entry). At this point the receiver vCPU can process the IPI, execute the TLB flush operation, and acknowledge the sender vCPU. This entire sequence must be repeated for each receiver vCPU affected by the invalidation of TLB entries.

The more the number of vCPUs, the longer the sender vCPU must wait for all acknowledgments to arrive. Hence, large-vCPU VMs experience a significant performance loss when running memory-intensive multi-threaded applications that frequently invalidate their virtual memory mappings. For instance, the Dedup benchmark in the Parsec benchmark suite [32] frequently invokes the `madvise()` system call to release address mappings that are no longer needed.

Our solution to this problem is to para-virtualize the TLB flush invocation in the guest. We modified the KVM hypervisor and the guest kernel to replace the emulation of TLB flush IPI with a more efficient hypercall-driven mechanism. Specifically, whenever TLB entries need to be invalidated, the sender vCPU invokes a hypercall requesting the hypervisor to directly flush the TLB, instead of generating an IPI for receiver VCPUs. The hypervisor then directly invalidates the affected TLB mappings on all pCPUs using native IPI delivery in the execution hardware, as opposed to the earlier described method of injecting virtual IPIs for receiver vCPUs to process in the guest mode. Since there are far fewer pCPUs than vC-

PUs, the sender vCPU only needs to wait until all the pCPUs invalidate their TLB entries for that VM and acknowledge the sender's pCPU. This optimization significantly speeds up TLB flush operation for large-vCPU VMs.

*Guest Modifications:* TLB flush mechanism in the guest kernel creates a CPU map of all the CPUs whose TLB needs to be invalidated. Sender vCPU first invalidates its TLB, then using the CPU map it sends TLB flush IPI to all other vCPUs. Our TLB patch intercepts this mechanism before it sends the IPI, instead, it makes a hypercall to the host. In this way, hypervisor handles the TLB flush for the respective VM without the need for TLB flush IPIs.

*Host Modifications:* On the host side, we add a hypercall handler for processing TLB flush. After KVM receives the hypercall it calls our handler with the information about the caller vCPU. Our handler uses a KVM-provided function to flush TLB entries for all vCPUs of the VM.

### C. Fast Reactivation of Idle vCPUs

To deliver a rescheduling IPI to an idle receiver vCPU, the sender vCPU writes required information to an MSR register [33] which causes a VMExit on the sender vCPU. Because of the VMExit, sender vCPU exits from the guest mode and KVM hypervisor takes control to handle the VM Exit. The exit handler first posts the IPI to the receiver vCPU and marks the vCPU thread ready for scheduling by the host CPU scheduler. Additionally, to reduce the time spent waiting in ready queue, the exit handler temporarily boosts (increases) the scheduling priority of the receiver vCPU. Specifically, instead of only making the vCPU ready to run, the receiver vCPU thread is also marked as the immediate next task to be scheduled on its pCPU. This ensures the receiver vCPU will be reactivated from its idle state without any scheduling delays and can immediately run the pinned application thread.

## V. Evaluation

In this section, we demonstrate that our techniques overcome key virtualization overheads when running parallel applications within large-VCPU VMs. We evaluate the performance of various parallel applications from the PARSEC benchmark suite [32] and demonstrate that large-vCPU VMs do not need to incur additional virtualization overheads with the right set of guest and host OS optimizations.

### A. Experimental Setup

Our test environment consists of an Intel Xeon server with 2.1GHz dual 6-core processors and 128 GB memory. To avoid interference from second-order effects, we disable hyper-threading and the second 6-core CPU in the BIOS settings. Thus the host machine effectively uses only 6 physical cores in all experiments. The VM is assigned 8 GB memory with varying number of vCPUs from 1 to 255 (the maximum vCPU count supported by KVM). Both the host and the VM run Linux kernel version 4.12.9 and QEMU version 2.5.0. Each data value is computed as an average over at least five experimental runs and preceded by a warmup run to eliminate the impact of cold caches and TLB. Where standard deviation is high, we repeat the experiments to increase confidence in our results.

Table I lists the six experimental configurations that we compare in our evaluations. In all configurations except those involving *fast vCPU wakeup*, the host OS runs an unmodified Completely Fair Scheduler (CFS), which is the default CPU scheduler in the Linux kernel. In the *Baseline* configuration, the VM runs an unmodified CFS scheduler. In the *pvTLB* configuration, the guest OS is modified to use our para-virtualized TLB flush mechanism when all online vCPUs need to flush their TLBs in response to changes in page-table entries. In the *Fast Wakeup* configuration, the hypervisor is modified to boost the priority of a vCPU that's being woken up from idle mode to run a guest task. In the *Per-Thread vCPUs* configuration, the guest OS is modified to pin one application thread to each vCPU, so that thread migration across vCPUs and the resulting inter-vCPU IPIs are eliminated. The last two configurations consist of combinations of the first four and help study incremental performance impact of combining different techniques.

### B. PARSEC Benchmark Suite

Princeton Application Repository for Shared-Memory Computers (PARSEC) [32] is a benchmark suite for multi-threaded applications. We discuss three applications from this suite in detail, namely, *Dedup*, *Vips*, and *Canneal*, for results presented in this paper. Results from other applications in the suite are discussed briefly due to space constraints. Dedup [32] compresses a data stream with a combination of global and local deduplication, and represents an I/O and CPU intensive parallel application. The VASARI Image Processing System (VIPS) [32] is a CPU and memory-intensive parallel application for image processing operations such as affine transformations and convolution. Canneal [32] is another CPU and memory-intensive parallel application that uses cache-aware simulated annealing to minimize the routing cost of chip design. Together, these applications allow us to stress test the effectiveness of different optimizations. We ran 255 threads for all the benchmarks, except for Blackscholes, which runs 128 threads because it requires thread count to be a power of 2.

### C. Application Execution Time

Figures 2, 3, and 4 show the execution times for Canneal, Vips, and Dedup, respectively, *when each of our techniques is used alone with CFS* compared to the CFS-only Baseline configuration. We first vary the number of vCPUs assigned to a VM from 1 to 6, followed by 50, 100, 150, 200 and 255 vCPUs. The Baseline configuration performs best with 6 vCPUs, when the degree of parallelism is the highest and contention is minimized. With more than 6 vCPUs, the execution time of the Baseline configuration worsens for all three benchmarks applications, but more for Vips and Dedup than Canneal.

| Configuration | Default CFS scheduler | Para-virtual TLB Flush | Fast vCPU Wakeup | Per-thread vCPUs |
|---|---|---|---|---|
| Baseline | X | | | |
| pvTLB | X | X | | |
| Fast Wakeup | X | | X | |
| Per-thread vCPUs | X | | | X |
| pvTLB + Fast Wakeup | X | X | X | |
| pvTLB + Fast Wakeup + Per-thread vCPUs | X | X | X | X |

TABLE I
VARIOUS COMBINATION OF CONFIGURATIONS USED IN EXPERIMENTS.

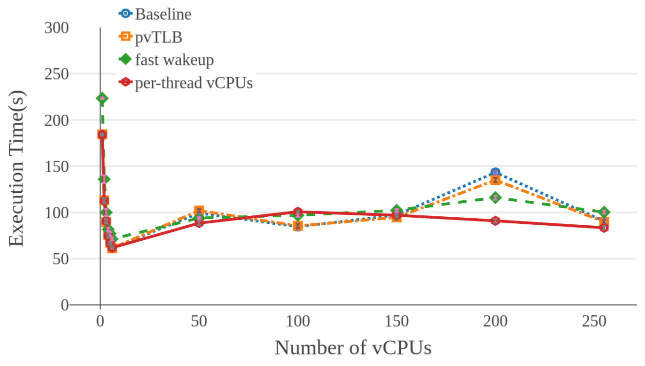

Fig. 2. Canneal: Effect of individual techniques on execution time.

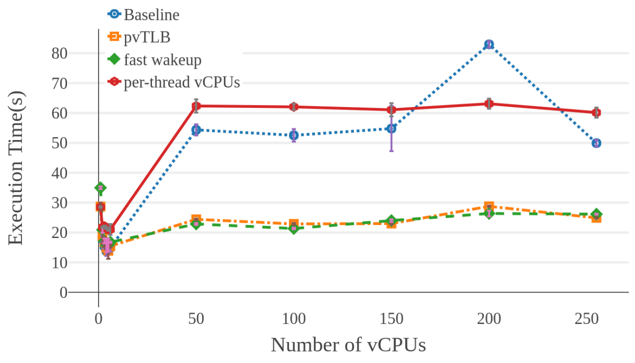

Fig. 4. Dedup: Effect of individual techniques on execution time.

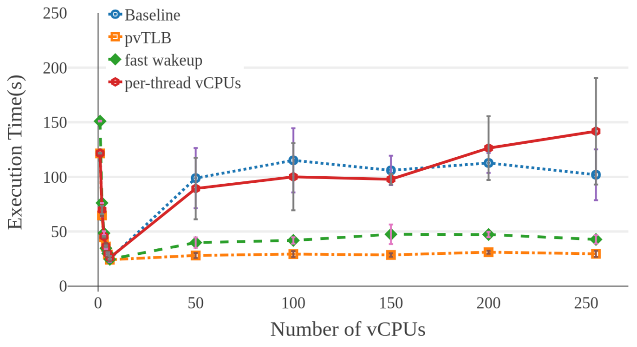

Fig. 3. Vips: Effect of individual techniques on execution time.

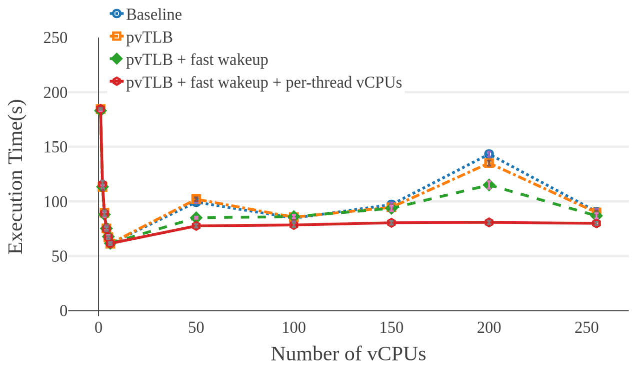

Fig. 5. Canneal: Effect of combining different techniques on execution time.

For Canneal, in Figure 2, the per-thread vCPUs configuration provides a slight performance improvement over baseline as the number of vCPUs exceeds 150. This improvement is because pinning threads to vCPUs prevents the guest OS from migrating threads among vCPUs for load balancing and eliminates the corresponding rescheduling IPIs. The other two techniques do not show any significant performance improvement over Baseline, since the overhead of TLB flushes and idle thread activation is small on Canneal runtime.

For Vips, in Figure 3, and Dedup, in Figure 4, fast wakeup and pvTLB configurations provide a large improvement over Baseline, since these applications have significant instances of idle vCPUs and TLB Flush IPIs. However, per-thread vCPUs optimization does not provide much performance gain over Baseline and performs slightly worse by preventing thread migration across vCPUs in the guest.

Next, we examine the performance when combinations of these techniques are used together. For Canneal, in Figure 5, the combination of all three techniques improves performance over Baseline by a larger margin than using per-thread vCPUs or pvTLB alone in Figure 2. Likewise, for Vips in Figure 6 and Dedup in Figure 7, combination of all three techniques performs better than Baseline by a large margin. More importantly, the combined use of all techniques matches the peak performance of Baseline configuration when using 6 vCPUs, and does not decrease as the number of vCPUs is increased to 255. Thus, the performance of a large-vCPU VM is not impacted by virtualization overheads.

Next, we examine in greater depth, the relationship of TLB Flush and Reschedule IPIs on these benchmark execution times.

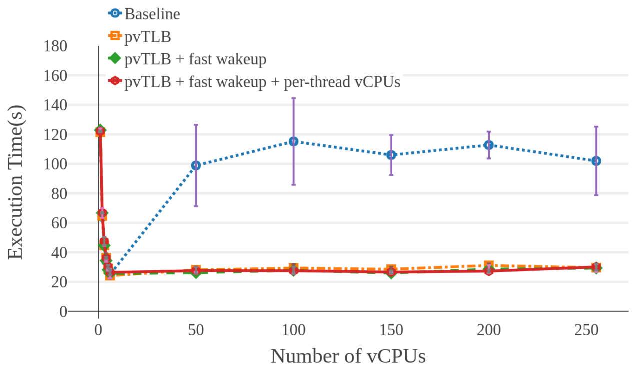

Fig. 6. Vips: Effect of combining different techniques on execution time.

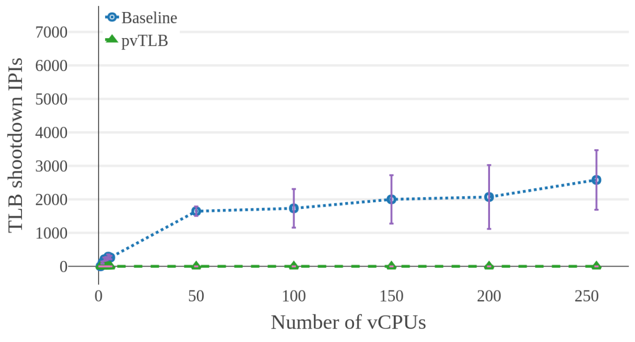

Fig. 8. Canneal: effect of number of vCPUs on TLB flush IPIs.

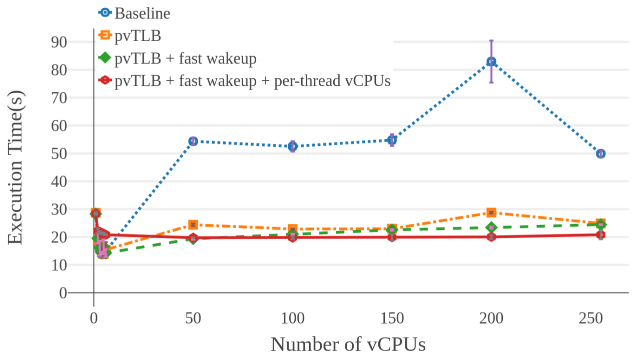

Fig. 7. Dedup: Effect of combining different techniques on execution time.

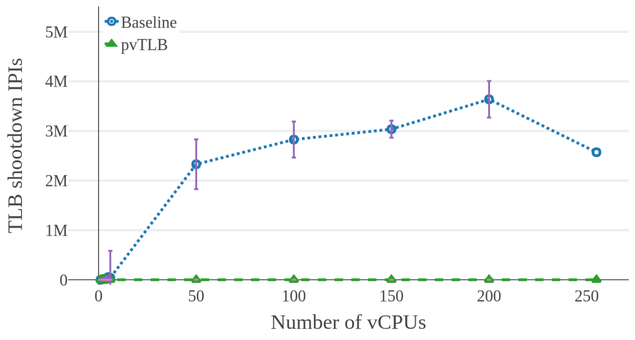

Fig. 9. Vips: effect of number of vCPUs on TLB flush IPIs.

### D. Addressing TLB Flush IPI Overhead

We discussed earlier that delivery of TLB Flush IPIs across vCPUs must be emulated by the hypervisor. More the TLB Flush IPIs, more the hypervisor-level emulation overhead. Here we study the impact of TLB Flush IPIs on application execution times discussed in the previous section.

Figures 8, 9 and 10 compare the number of TLB Flush IPIs in the Baseline and pvTLB configurations, as the number of vCPUs is increased for Canneal, Vips, and Dedup applications respectively. Specifically, for Baseline configuration, we observe that increasing the number of vCPUs in a VM also increases the number of TLB Flush IPIs. In contrast, the pvTLB configuration replaces IPI-driven TLB flush with a para-virtualized TLB Flush IPI, hence we do not see any TLB Flush IPIs in the guest.

Next, let's correlate these TLB Flush IPI counts with the execution times seen earlier. We see that, for Vips in Figure 3 and Dedup in Figure 4, both the Baseline execution times and its reduction using pvTLB correlate closely with the respective IPI counts in Figures 9 and 10. Vips and Dedup are both memory-intensive applications which trigger large number of TLB Flush IPIs, hence para-virtualizing TLB Flush operation produces a major difference in application performance.

On the other hand, for Canneal, the execution time in Figure 2 does not show a significant improvement from the use of pvTLB optimization. We observe from Figure 8 that, relative to Vips and Dedup, Canneal triggers very few TLB Flush IPIs. Hence its performance gains from pvTLB optimization is also small.

### E. Addressing Overhead in Reactivation of Idle vCPUs

Here we study the impact of overhead in reactivating idle vCPUs on application execution times discussed in the Section V-C. As discussed earlier in design, idle vCPUs transition to hypervisor mode, where the hypervisor places the corresponding host-level vCPU thread in a blocked state. The vCPU thread remains blocked till it has a guest task ready to execute and is woken up by a rescheduling IPI delivered from a peer vCPU. The delivery of this inter-vCPU rescheduling IPI is emulated in software by the hypervisor and introduces delays in reactivating and idle-mode vCPU. We expect that the number of rescheduling IPIs to increase with the number of vCPUs because the likelihood of a vCPU being idle also increased. To reduce this overhead, we modified the rescheduling IPI delivery mechanism to temporarily boost the scheduling priority of the receiver vCPU thread by designating the IPI receiver as the immediate next task to be scheduled by host-level CPU scheduler.

Figures 11, 12 and 13 show the increase in the number of rescheduling IPIs with increase in the number of vCPUs. We notice that thread pinning mechanism in per-thread vCPUs triggers even more rescheduling IPIs for Canneal and Dedup.

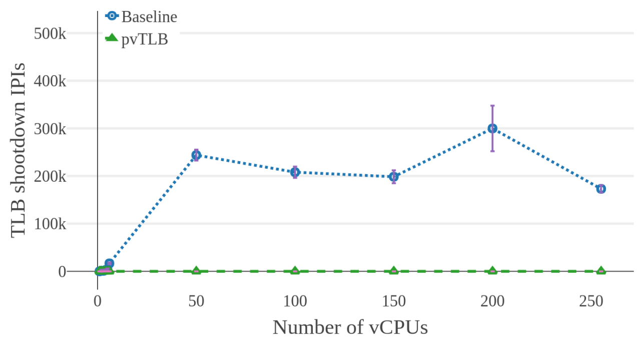

Fig. 10. Dedup: effect of number of vCPUs on TLB flush IPIs.

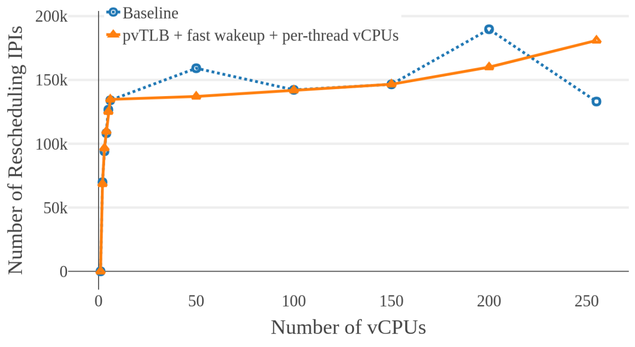

Fig. 12. Vips: effect of number of vCPUs on rescheduling IPIs.

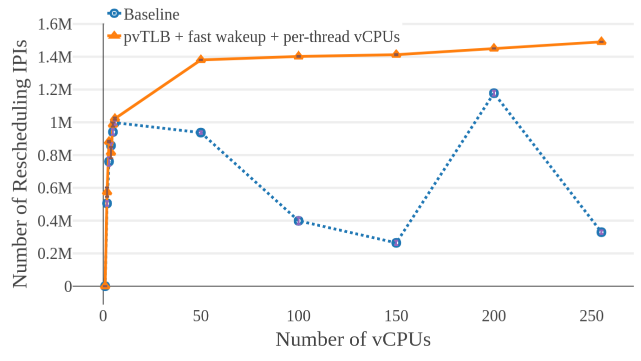

Fig. 11. Canneal: effect of number of vCPUs on rescheduling IPIs.

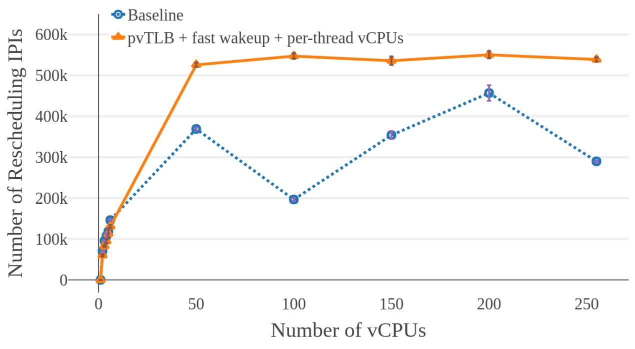

Fig. 13. Dedup: effect of number of vCPUs on rescheduling IPIs.

With unmodified CFS kernel in the guest, if a guest task is woken up, an idle but scheduled vCPU could steal the woken task from another pre-empted vCPU, instead of sending a rescheduling IPI. But with pinning mechanism, a rescheduling IPI must be sent to the vCPU where the task is pinned.

Ironically, while using pinning mechanism helps us to eliminate guest side scheduling it also causes more rescheduling IPIs. However, since we accelerate the delivery of rescheduling IPI and reactivation of idle vCPUs, the net impact of additional IPIs is minimal on execution time.

Figures 2, 3 and 4 show that our Fast Wake-up is able to reduce the overhead due to delays in reactivating idle vCPUs for all three applications, even when there are a high number of rescheduling IPIs. Especially for Canneal in Figure 5, it can be seen that performance improves a bit with the ”pvTLB + fast wakeup” configuration and we know from Figure 11 that Canneal does not generate too many TLB Flush IPIs but generates significant rescheduling IPIs.

To get a better understanding we also examined VM Exit reasons in the hypervisor and found that the majority of the VM Exits were due to the MSR Write operations, which triggers IPI via x2APIC [31]. We verified the number of rescheduling IPIs correlate with MSR Write related VM Exits.

## F. CPU Utilization

We not only looked at the execution time but also examined CPU utilization overhead due to our techniques compared to the Baseline configuration. Figures 14, 15 and 16 show the CPU utilization for three benchmark applications. Figures 14 and 15 show that, in the case of Canneal and Vips while CPU utilization is comparable to Baseline across all vCPU counts, the execution time improves in Figures 5 and 6. In case of Dedup we actually observe a slight reduction in CPU utilization in Figure 16 while simultaneously improving the performance in Figure 7.

## G. Adaptation to Different pCPU Configurations

We also show that our VM can adapt to a different physical CPU counts after a VM migration without changing the vCPU configuration, such as having to rely on hotplug mechanisms to reconfigure a VM. We conducted experiments in which we varied the number of physical CPUs. For these experiments we changed the number of pCPUs from 1 to 12. Table II lists the four configurations we used.

Our reference for comparison, called Baseline-equal, is the original Linux CPU scheduler in a VM that has the same number of vCPUs as the pCPUs. The Baseline-255 configuration is meant to examine how the Linux scheduler would perform when it is configured with the maximum allowed

| Configuration | Explanation |
|---|---|
| Baseline-equal | Baseline with vCPUs = pCPUs |
| pvTLB + fast wakeup + per-thread vCPUs-equal | pvTLB + fast wakeup + per-thread vCPUs with vCPUs = pCPUs |
| Baseline-255 | Baseline with vCPUs = 255 |
| pvTLB + fast wakeup + per-thread vCPUs-255 | pvTLB + fast wakeup + per-thread vCPUs with vCPUs = 255 |

TABLE II
CONFIGURATIONS FOR STUDYING THE EFFECT OF PCPUS VARIATION.

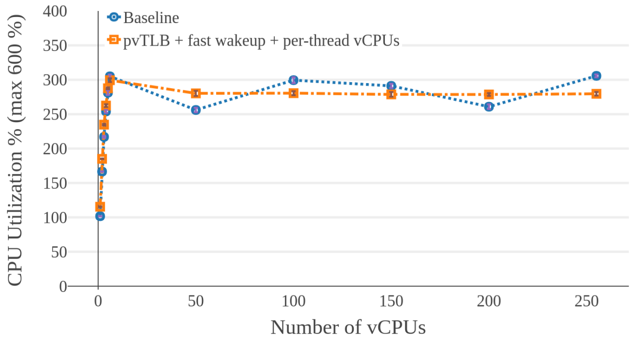

Fig. 14. Canneal: Host CPU Utilization for 6 pCPUs.

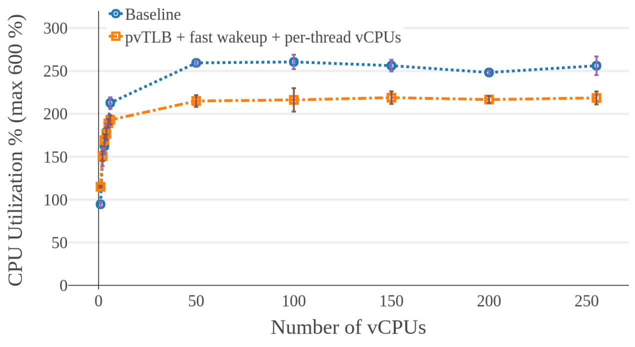

Fig. 16. Dedup: Host CPU Utilization for for 6 pCPUs.

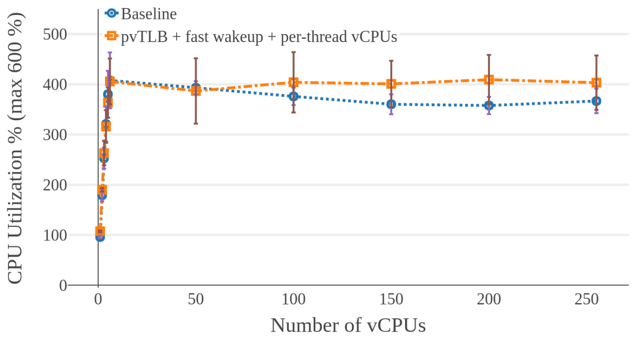

Fig. 15. Vips: Host CPU Utilization for 6 pCPUs.

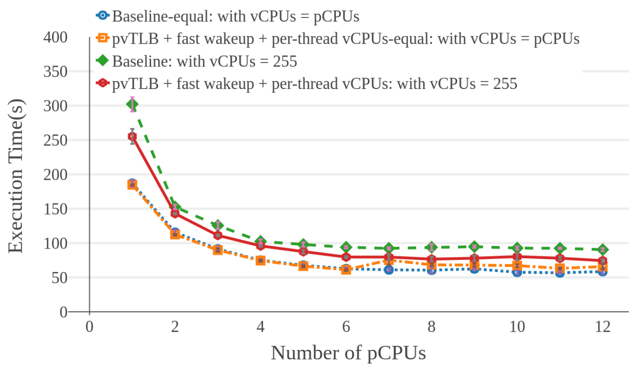

Fig. 17. Canneal: Performance in VM with varying pCPUs.

255 vCPUs. The ”pvTLB + fast wakeup + per-thread vCPUs-equal” configuration allows us to examine the performance when the number of vCPUs equals the number of pCPUs. The ”pvTLB + fast wakeup + per-thread vCPUs-255” configuration allows us to examine how our techniques adapt to the changing the number of pCPUs when the VM has 200 vCPUs.

Figure 17, Figures 18 and 19 show that our ”pvTLB + fast wakeup + per-thread vCPUs-equal” configuration matches the peak performance of Baseline-equal. More importantly, ”pvTLB + fast wakeup + per-thread vCPUs-255” outperforms ”Baseline-255” due to reduction of key processor virtualization overheads, and approaches Baseline-equal performance for larger pCPU counts.

*These results show that, when the vCPU count exceeds pCPU count, our techniques help reduce inter-processor synchronization overheads. Also, after a VM migration to a machine with additional physical CPU cores, our techniques allow a VM to benefit from additional parallelism having to reconfigure additional vCPUs.*

### H. Other Applications in PARSEC Suite

We also evaluated other applications from PARSEC benchmark suite for the effectiveness of our techniques. Figures 20, 21, 22 and 23 compare our techniques against Baseline for Blackscholes, Bodytrack, Ferret, and Raytrace applications. As can be seen, the Baseline CFS kernel scales well with vCPU count for all four benchmarks (except for BodyTrack when using 200 vCPUs in Figure 21). Our technique closely matches or improves upon the Baseline performance for all benchmarks.

## VI. RELATED WORK

In this section, we contrast our work against prior approaches to address performance overheads in processor scheduling for VMs. Prior approaches can be categorized as addressing *vCPU synchronization overheads* and *double scheduling* problems.

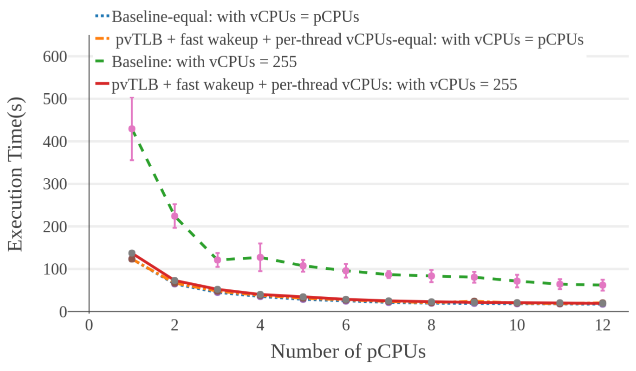

Fig. 18. Vips: Performance in VM with varying pCPUs.

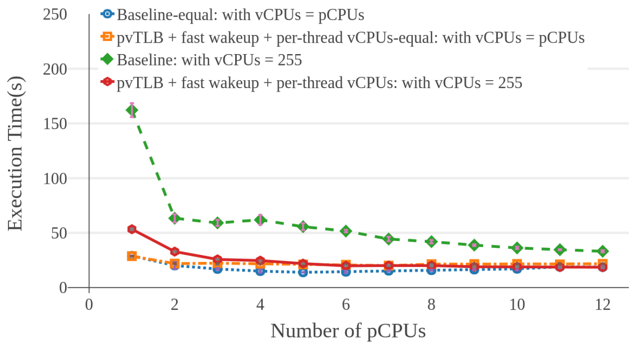

Fig. 19. Dedup: Performance in VM with varying pCPUs.

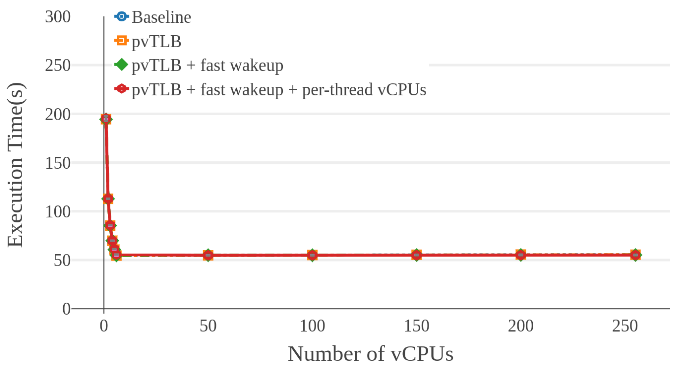

Fig. 20. Blackscholes: Effect of combining different techniques on execution time.

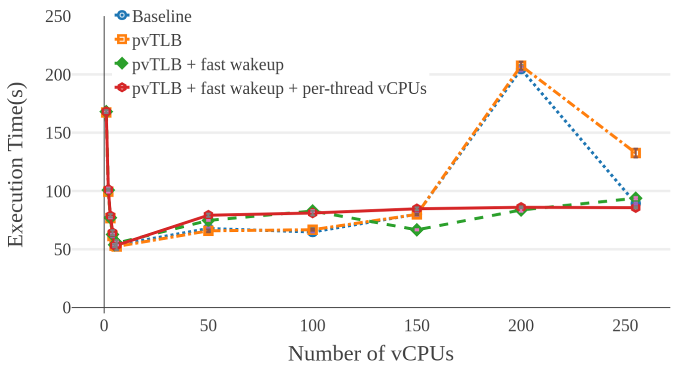

Fig. 21. Bodytrack: Effect of combining different techniques on execution time.

## A. *Eliminating Double Scheduling*

The double scheduling problem refers to scheduling inefficiencies that result from independent scheduling decisions made by the CPU schedulers inside the guest OS and the host OS. FlexCore [19], [20] advocates the position that the CPU scheduling in the host OS must be simplified by partitioning available pCPUs among all active VMs. Depending on guest workload, the number of vCPUs seen by each VM is dynamically adjusted to match the number of pCPUs assigned to a VM, using hot plugging/unplugging mechanism. All vCPUs of a VM are scheduled simultaneously, via Gang Scheduling, which eliminates delays in IPI delivery and the lock-holder pre-emption problem. However, the number of active VMs hosted by the system cannot exceed the total number of pCPU cores, which limits consolidation efficiency. Another work, called vScale [21], eliminates the limitation in FlexCore by allowing pCPUs to be shared by more than one vCPU. However, both FlexCore and vScale rely on dynamically adding or removing guest vCPUs using either hypervisor-level hot plugging/unplugging [22] or guest-level online/offline mechanisms. These mechanisms cannot be frequently invoked, since they are heavyweight and disruptive to guest operations; the guest OS needs to account for vCPUs that suddenly appear and disappear by reconfiguring its internal structures every time and migrating active processes between vCPUs. In contrast, our approach eliminates the need to guess or keep changing the number of vCPUs; we advocate the opposite solution where each VM gets maximum possible vCPUs which remains unchanged for the life of the VM; each application thread in the VM gets its own dedicated vCPU, eliminating the need for most CPU scheduling in the guest OS.

## B. *Reducing Inter-vCPU Synchronization Cost*

Prior approaches have also focused on reducing the overhead of emulating synchronization events between vCPUs, such as TLB flush requests and spinlock synchronization across vCPUs. IPI-driven co-scheduling [14] tries to schedule together IPI sender and receiver vCPUs. Partial co-scheduling [15] improves host utilization compared to strict co-scheduling by detecting related vCPUs that communicate with each other through shared pages. KVM [17] provides a paravirtualized remote flush TLB mechanism so that sender vCPU does not need to wait for acknowledgment from a pre-empted or sleeping vCPU. Although it reduces the latency of TLB flush operations, it does not eliminate the delivery overhead of TLB flush IPI. Shoot4U [18] proposes a paravirtualized TLB flush operation which, like our work, also emulates TLB flush operation in the hypervisor, without injecting a function call IPI into the guest. The flush handler in the

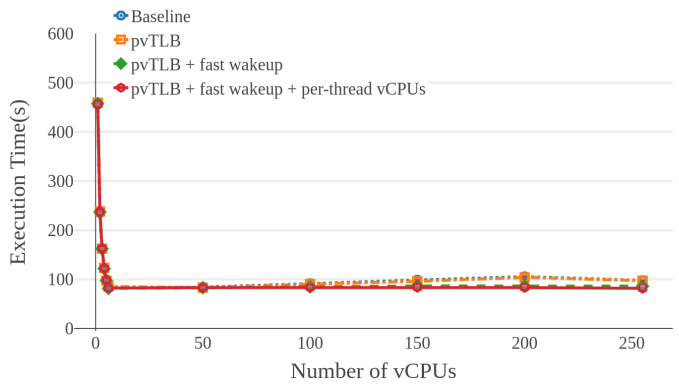

Fig. 22. Ferret: Effect of combining different techniques on execution time.

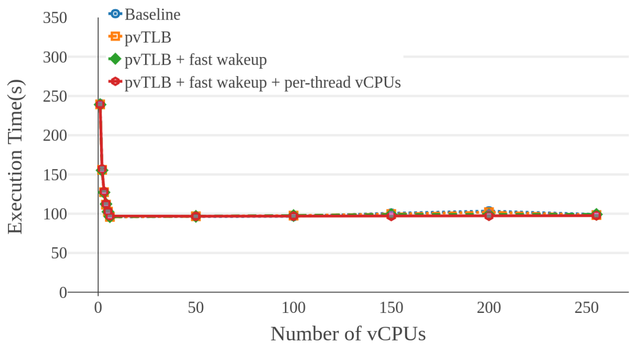

Fig. 23. Raytrace: Effect of combining different techniques on execution time.

hypervisor iterates over all vCPUs of the guest, invalidating relevant address mappings. While this approach works well for small number of vCPUs, the overhead increases rapidly for large-vCPU VMs. In our work, we directly invalidate TLB entries on pCPUs, instead of vCPUs, thus bounding the TLB flush cost in proportion to pCPUs count, even when the vCPU count is high.

Uhlig et al. [11] proposed paravirtualized and fully virtualized solutions to reduce the busy waiting time over spin locks. Fribel et al. [34] proposed techniques to detect busy waits of vCPU in the hypervisor. Para-virtualized ticket spinlock [12], [13] uses a FIFO queue to prioritize vCPU scheduling order. While these techniques are useful, we use a simpler existing KVM support for Pause-Loop Exiting (PLE) [35] to have a spinning vCPU exit to hypervisor earlier, so that the lock holding vCPU gets a chance to complete its critical section.

Finally, unlike the above approaches, our work also speeds up the reactivation of idle vCPUs to handle new threads in response to rescheduling IPI from another vCPU.

## VII. Conclusion

Parallel applications running in VMs suffer from degraded performance when the number of virtual processors (vCPUs) exceeds the number of physical processors (pCPUs). In this paper, we tracked the root causes of this performance degradation as being due to the increased cost of emulating inter-vCPU synchronization events, such as TLB flushes and rescheduling IPIs. To efficiently run large-vCPU VMs with low hypervisor-induced overheads, we presented three techniques to effectively decouple parallel application performance from the number of vCPUs. The first technique dedicates a vCPU to run each guest thread to eliminate IPIs triggered by inter-vCPU work stealing. The second technique para-virtualizes the emulation of inter-vCPU TLB flushes. The third technique speeds up the reactivation of idle vCPUs by prioritizing the delivery of rescheduling IPIs. We evaluated a prototype of our techniques in the Linux KVM/QEMU platform using several parallel benchmarks and showed that their performance is sustained even when the vCPU count far exceeds the pCPU count. Our techniques enable support for large-vCPU VMs on hosts with fewer pCPUs and also exploit increased parallelism when VMs are migrated to hosts with more pCPUs.

## Acknowledgment

This work is funded in part by the National Science Foundation through award CNS-1527338.

## References

[1] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007, pp. 225–230.

[2] 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," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[3] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 8–11, 2006.

[4] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, "Analysis of virtualization technologies for high performance computing environments," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2011.

[5] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Cloud Computing Technology and Science (CloudCom)*, 2010, pp. 409–416.

[6] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for HPC systems," in *Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops*. Springer Berlin Heidelberg, 2006, pp. 474–486.

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

[8] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed I/O." *Intel technology journal*, vol. 10, no. 3, 2006.

[9] AMD Corporation, "AMD Virtualization (AMD-V) technology https://www.amd.com/en-us/solutions/servers/virtualization."

[10] L. Lamport, "On interprocess communication," *Distributed computing*, vol. 1, no. 2, pp. 86–101, 1986.

[11] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski, "Towards scalable multiprocessor virtual machines." in *Virtual Machine Research and Technology Symposium*, 2004, pp. 43–56.

[12] K. Raghavendra, "Paravirtualized ticket spinlocks https://lwn.net/articles/495597/."

[13] S. Kashyap, C. Min, and T. Kim, "Opportunistic Spinlocks: Achieving virtual machine scalability in the clouds," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 1, pp. 9–16, 2016.

[14] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng, "Demand-based coordinated scheduling for SMP VMs," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 369–380, 2013.

[15] A. Busse, J. H. Schönherr, M. Diener, P. O. Navaux, and H.-U. Heiß, "Partial coscheduling of virtual machines based on memory access patterns," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015.

[16] O. Sukwong and H. S. Kim, "Is co-scheduling too expensive for SMP VMs?" in *Proceedings of the European Conference on Computer Systems (Eurosys)*, 2011, pp. 257–272.

[17] N. A. Dadhania, "Kvm paravirt remote flush TLB https://lwn.net/articles/500188/."

[18] J. Ouyang, J. R. Lange, and H. Zheng, "Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted vCPUs," in *Proceedings of the 12th International Conference on Virtual Execution Environments (VEE)*, 2016, pp. 17–23.

[19] T. Miao and H. Chen, "FlexCore: Dynamic virtual machine scheduling using VCPU ballooning," *Tsinghua Science and Technology*, vol. 20, no. 1, pp. 7–16, 2015.

[20] X. Song, J. Shi, H. Chen, and B. Zang, "Schedule processes, not VCPUs," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 2013.

[21] L. Cheng, J. Rao, and F. Lau, "vScale: Automatic and efficient processor scaling for smp virtual machines," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[22] Z. Mwaikambo, A. Raj, R. Russell, J. Schopp, and S. Vaddagiri, "Linux kernel hotplug CPU support," in *Linux Symposium*, 2004.

[23] S. Panneerselvam and M. M. Swift, "Chameleon: Operating system support for dynamic processors," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 99–110, 2012.

[24] S. Panneerselvam, M. M. Swift, and N. S. Kim, "Bolt: Faster reconfiguration in operating systems." in *USENIX Annual Technical Conference*, 2015, pp. 511–516.

[25] R. Harper and K. Rister, "KVM limits arbitrary or architectural?" in *KVM Forum*, 2008.

[26] J. Nakajima, "Enabling optimized interrupt/APIC virtualization in KVM," in *KVM Forum*, 2012.

[27] I. Ahmad, A. Gulati, and A. Mashtizadeh, "vIC: Interrupt coalescing for virtual machine storage device I/O," in *USENIX Annual Technical Conference*, 2011.

[28] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Hot Topics in Operating Systems*, 2001.

[29] C. Lameter, "Effective synchronization on Linux/NUMA systems," in *Gelato Conference*, 2005.

[30] Linux Kernel Documentation, "Cpusets: https://www.kernel.org/doc/documentation/cgroup-v1/cpusets.txt."

[31] Intel Corporation, "Intel 64 architecture x2APIC specification https://software.intel.com/en-us/download/intel-64-architecture-x2apic-specification."

[32] P. B. Suite, "http://parsec.cs.princeton.edu."

[33] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization." *Intel Technology Journal*, vol. 10, no. 3, 2006.

[34] T. Friebel and S. Biemueller, "How to deal with lock holder preemption," in *Xen Summit North America*, 2008.

[35] Rik van Riel, "Directed yield for pause loop exiting. https://lwn.net/articles/424960/."