Characterizing and Optimizing the Performance of Multithreaded Programs Under Interference

Yong Zhao Jia Rao
The University of Texas at Arlington
yong.zhao@mavs.uta.edu
jia.rao@uta.edu

Qing Yi
The University of Colorado Colorado Springs
qyi@uccs.edu

ABSTRACT

As virtualization becomes ubiquitous in datacenters, there is a growing interest in characterizing application performance in multi-tenant environments to improve datacenter resource management. The performance of parallel programs is notoriously difficult to reason about in virtualized environments. Although performance degradations caused by virtualization and interferences have been extensively studied, there still lacks a comprehensive understanding why parallel programs have unpredictable slowdowns when co-located with different types of workloads.

This paper presents a systematic and quantitative study of multithreaded performance under interference. We design synthetic workloads to emulate different types of interference and study the behavior of parallel programs under such interferences. We find that unpredictable performance is the result of complex interplays between the design of the program, the memory hierarchy of the host system, and the CPU scheduling at the hypervisor. To understand the intricate relationships between multiple factors, we decompose parallel runtime into compute, synchronization and steal time, and use the runtime breakdown to measure program progress and identify execution inefficiency under interference. Based on these findings, we develop an online approach to predicting performance slowdown without requiring parallel programs to be completed, and devise two scheduling optimizations at the hypervisor to reduce slowdowns. Experimental results with Xen and representative parallel workloads show that the online performance prediction achieves on average less than 4.5% error and the optimizations reduce runtime slowdown by as much as 38% compared to stock Xen.

Keywords

Virtual Machine Scheduling; Multicore Systems; Performance Modeling; Parallel Program Optimization

1. INTRODUCTION

PACT '16, September 11-15, 2016, Haifa, Israel

DOI: http://dx.doi.org/10.1145/2967938.2967939

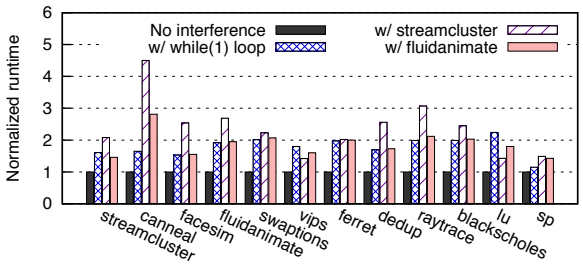

Figure 1: The runtime slowdown of parallel workloads under different interference scenarios.

Cloud computing, powered by warehouse-scale datacenters, provides users with abundant parallelism and potentially unlimited scalability. While the cloud is believed to be an ideal platform for hosting parallel applications, its nature of multi-user sharing and resource over-commitment makes parallel performance often quite disappointing and unpredictable. Although many studies [11, 16, 23, 29, 32, 33] have identified the excessive synchronization delays due to multi-tenant interferences as the culprit, there lacks a full understanding of the quantitative relationship between changes in synchronization and the overall performance loss. As performance modeling plays a fundamental role in designing traditional parallel systems, a systematic and quantitative study of parallel performance under cloud interferences would help improve the resource and power management in datacenters. Most importantly, predicting parallel performance allows users to evaluate the outcome of their cloud lease without completing long-running applications and select cloud services wisely to meet their expectations.

However, parallel performance is notoriously difficult to reason about in a shared cloud environment. Memory locality and resource contentions on the CPU function units, shared cache and memory bandwidth could all affect the speed of individual threads. Moreover, the performance of parallel programs as a whole also depends critically on the cooperation of multiple threads. Contentions on CPU time cause the well-known lock-holder preemption (LHP) problem [32], in which threads holding important locks are prematurely de-scheduled, leading to exceedingly long synchronizations and out-of-sync executions at multiple threads. The major challenges in modeling parallel performance are twofold: 1) individual threads have varying slowdowns when co-located with different interferences; 2) the quantitative relationship between slowdowns at individual threads and the overall slowdown remains unclear.

To demonstrate the severity of performance degradation and its unpredictability in multi-tenant clouds, Figure 1 shows the slowdown of various parallel programs under different interference scenarios. The parallel workloads include programs from the PARSEC [31] and NASA parallel benchmarks (NPB) [2] with different synchronization methods (i.e., busy-waiting and blocking), varying task granularities, and different work assignment policies (i.e., static and dynamic/work-stealing assignments). The interferences contain a synthetic workload that only competes for CPU cycles (i.e., `while(1)` loop) and two real workloads from the PARSEC benchmarks. Both the parallel workloads and the interferences run with 4 threads (see Section 3.1 for detailed settings). From the figure, we can see that performance slowdowns vary substantially across different workloads, even under the same type of interference. For example, the synthetic loop incurred more than 100% slowdown to *lu* while it merely slowed *sp* by 13%. More interestingly, slowdowns become more unpredictable when the background interference changes. For example, while *facesim* suffers a 165% slowdown with *streamcluster*, its degradation with *fluidanimate* is only 53%. In contrast, some workloads suffer more severe slowdowns under *fluidanimate* compared to that under *streamcluster*, e.g., *lu* and *vips*. The difference in memory contentiousness of the interfering workloads alone can not explain the varying slowdowns. The unpredictability is the result of complex interplays between the designs of the program, the memory hierarchy of the host machine, and the underlying CPU scheduling at the hypervisor. Understanding how these factors interact with each other is crucial to fully understanding parallel performance in the cloud.

In this paper, we aim to uncover the mystery of parallel performance under interference. Our methodology is to study parallel performance in controlled experiments so that individual factors can be examined separately. To this end, we design a set of synthetic workloads to emulate different types of interference. The synthetic workloads include `while(1)` loops with *persistent*, *periodic* and *intermittent* interference patterns and configurable CPU demands. To further pinpoint the source of unpredictability, we design a profiling tool, `vProfile`, to derive a detailed breakdown of parallel runtime, and to report important scheduling and hardware statistics. With the help of fine-grained profiling, we have the following findings: 1) the slowdown of individual threads depends on the amount of CPU allocated to the thread, which is largely affected by the synchronization granularity of the program in a shared environment; 2) while interference is believed to slow down individual threads due to contentions on shared resources, it can constructively accelerate threads in memory-bound applications due to the alleviation of intra-program contentions; 3) Uncoordinated scheduling of co-running parallel programs causes out-of-sync execution between multiple threads, leading to further performance degradation.

Based on these findings, we develop an online approach to predicting the performance of parallel programs under interference. Our online prediction first profiles parallel execution under interference for a short period of time and compares the completed *useful* work during sampling with that in a dedicated environment during the same length of period. The difference in completed useful work is used to predict the overall slowdown. We define useful work as the necessary computation needed by a thread assuming an ideal memory system with zero latency and perfect synchronization with no spinning and blocking cost. Specifically, we determine useful work by removing cycles that are stolen by other tenants, perform spinning or context switching (i.e., sleep and wakeup), and are spent in the memory hierarchy from the profiling sample. Experimental results show that, for regular parallel programs in which individual threads are assigned the same amount of work and perform iterated computations, the online approach achieves on average less than 4.5% errors in predicting the overall slowdown.

We also devise two scheduling optimizations at the hypervisor to reduce slowdowns. To avoid premature preemptions, we propose *delayed preemption* (DP) to interleave the computations of the parallel program and background interferences. We further make DP adaptive to the varying synchronization granularities for different programs. Experimental results show that DP improves the performance of PARSEC benchmarks by up to 23%. Another optimization motivated by our analysis is *differential scheduling* (DS), which purposely avoids co-scheduling of parallel threads by having different time slices on multiple CPUs. Results show that this simple technique outperforms stock Xen by as much as 38% in NPB benchmarks.

The rest of the paper is organized as follows. Section 2 introduces the design of vProfile and the synthetic interferences. Section 3 provides an in-depth analysis of parallel performance under different types of interference. Section 4 and 5 present an approach for online performance prediction and two hypervisor-level optimizations motivated by the analysis, respectively. Section 6 discusses limitations and future work. Section 7 discusses related work and Section 8 concludes this paper.

2. PROFILING THE PERFORMANCE OF MULTITHREADED PROGRAMS

The key to understanding parallel performance under interference is to identify the sources of slowdown. A breakdown of parallel runtime from experiments in a controlled environment would help pinpoint the culprit and inspire possible remedies. In general, parallel runtime consists of compute time and idle time. Traditionally, idle time due to load imbalance and synchronization is considered the major source of parallel overhead as no progress can be made during idle time. In a virtualized environment, guest operating systems (OSes) schedule application threads onto virtual CPUs (vCPUs) and multiple vCPUs from different virtual machines (VMs) can share the same physical CPU (pCPU). Steal time is the time a vCPU waits to run on a pCPU while the hypervisor is servicing another vCPU. A large steal time indicates severe contentions on the CPU allocation. Therefore, we decompose parallel runtime into *compute*, *synchronization* and *steal* time to study the causes of slowdown. We design `vProfile` to report the breakdown of parallel runtime and record important scheduling statistics in the hypervisor. vProfile provides two hypercalls [1]: `vprofile_start` and `vprofile_stop` to mark the start and end of a profiling period.

2.1 Decomposing Parallel Runtime

During profiling, vProfile tracks per-vCPU state changes

[1]We describe the design of vProfile in a Xen environment. Other hypervisors can be easily modified to support vProfile.

in the hypervisor. Xen defines four vCPUs states, i.e., `running`, `runnable`, `blocked`, and `offline`. Steal time can be accounted using time spent in the runnable state, which counts the time a vCPU is ready to run but fails to acquire the pCPU. The accounting of synchronization time depends on the synchronization methods used by the parallel program. For blocking synchronization, such as `mutex` and `semaphore`, synchronization time is simply the time a vCPU stays in the blocked state.

Accounting synchronization time is more challenging for programs using busy-waiting synchronization (e.g., spinlocks) as vCPUs are always in the running state. There is existing work detecting spinning by instrumenting guest OS kernels [33], tracking user-kernel mode switches [32], and monitoring hardware performance events [5, 25]. We use the lightweight spin detection proposed in [25] to break the time in the running state into compute and synchronization (or spinning) time. As spin loops usually contain only a few instructions and are executed repeatedly, spinning vCPUs show high branch per instruction and low branch miss prediction rates compared to sibling vCPUs that are performing regular computation. We add a new `spinning` state to Xen and place a vCPU to such a state when spinning is detected. Synchronization time is then the time a vCPU stays in the spinning state.

2.2 Recording Performance Events

The breakdown of execution time alone is not sufficient to identify the causes of slowdown. For example, an increase of synchronization could be due to long latencies at a few synchronization points or prolonged wait time at many places. Detailed execution statistics can help find the root cause and develop approaches to mitigate slowdowns. vProfile reports per-vCPU statistics of the scheduling and hardware performance events listed in Table 1. Events `YIELD` and `PREEMPT`

Event	Description
`YIELD`	Voluntary yield to other vCPUs due to idling
`PREEMPT`	Involuntary preemption by the scheduler
`IDLE`	The time the pCPU in the idle state
`HARDWARE_STAT`	Statistics from hardware performance counters

Table 1: vProfile performance events.

shed light on the contentions between co-located VMs while `IDLE` reflects the overall utilization of the pCPU as well as the efficiency of the scheduler. Performance statistics from hardware counters can trace the low-level program behaviors under interference and reveal the complex interactions between the program and the hardware. vProfile can be configured to track various hardware statistics, including cycles spent in the offcore memory system (`OFFCORE_STALL`), LLC misses per thousand instructions (`MPKI`), and other events related to cache coherence traffic between private L2 caches.

2.3 Enabling Dedicated Mode

vProfile enables online performance prediction by temporarily throttling co-located workloads to estimate their reference performance. It provides a short period of dedicated execution to emulate the performance on a dedicated machine. We assume that the sampling should cover a number of major iterations of the parallel program and contain sufficient information for performance prediction. The dedicated mode can be enabled multiple times once program phase change is detected. vProfile can integrate existing phase change detection techniques.

vProfile exports a new hypercall `sys_enable_dedicated` to the VM hosting the parallel application. Upon receiving the hypercall, the hypervisor takes vCPUs other than the calling VM's vCPUs off pCPUs' run queues and places them into the `offline` mode. The hypervisor freezes vCPUs of co-running VMs for a pre-defined period (e.g., 300 scheduling epochs). To minimize the impact of the dedicated mode on regular scheduling, we disable CPU time accounting (i.e., credit debiting in Xen's credit scheduler) during dedicated mode. As such, fair CPU allocation is not affected when normal execution is resumed.

2.4 Synthetic Interference

Synthetic workloads should slow down individual threads to faithfully reflect contentions on CPU time and shared resources, such as the last-level cache (LLC) and memory bandwidth. It should also be able to emulate the complex patterns of real workloads that simultaneously interfere with multiple threads. To this end, we design the synthetic workloads as simple CPU loops consisting of interleaved busy and idle intervals. The busy-to-idle ratio, which is configurable, determines the intensity of the interference. While the synthetic interference only contends for CPU cycles, it can emulate contentions on shared hardware resources because a decrease in allocated CPU time is equivalent to an increase in memory access cost for the parallel applications under test. We create the following three types of interference to study the complex interplay between the parallel program, the memory hierarchy and the underlying CPU scheduling:

- *Persistent* interference comprises of simple `while(1)` loops demanding 100% of CPU time. It emulates the CPU demand of long-running sequential jobs or parallel applications with busy-waiting synchronization. Due to its simplicity, it is scheduled by hypervisors at predictable time points and does not incur preemptions to the parallel threads.
- *Periodic* interference demands CPU at regular intervals or otherwise stays idle. The ratio of the CPU burst and the idle period determines the level of contention (i.e., CPU demand). Periodic interference has fixed burst-to-idle ratio and fixed length of computation at each interval. It emulates regular parallel applications that have predictable computations and synchronizations. Periodic interference is more complex than persistent interference as it sleeps and wakes up periodically, leading to preemptions of parallel threads.
- *Intermittent* interference demands CPU at irregular intervals. The ratio of CPU burst and sleep remains unchanged, but the length of computation changes randomly. It emulates multi-programmed workloads with independent (random) demands from individual threads or parallel applications with irregular CPU demands. Compared to periodic interference, whose computation and idling are predictable, intermittent interference has unpredictable demands. This helps to study the behavior of parallel programs when their execution is out-of-sync.

We use the method of differential analysis [18, 20] to compare the execution profiles of parallel programs under different types of interference. The low-level metrics that are

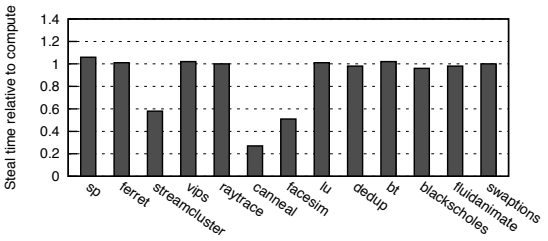

Figure 2: Varying resilience to interference.

highly correlated to the overall performance are examined to identify the causes of performance slowdown. We set the CPU demand of periodic and intermittent interference to 50% of the pCPU. The periodic interference performs 10ms computation and stays idle for 10ms. In contrast, the computation in intermittent interference varies randomly from 1ms to 40ms with the idle period changing accordingly to generate the 50% CPU demand. We use single-threaded interferences to study the slowdown of individual threads and the multi-threaded version to evaluate the efficiency of multiple parallel threads.

3. UNDERSTANDING PARALLEL PERFORMANCE

In this section, we use the statistics reported by vProfile to explain the mystery of parallel performance under interferences. We found that programs show different levels of resilience to interferences, leading to varying CPU allocations to the parallel program (Section 3.2). Interference may accelerate program execution by reducing the compute time needed in normal execution (Section 3.3). Finally, the overall performance is determined by the complex interplay between multiple factors (Section 3.4).

3.1 Experimental Setup

We ran shared-memory multi-threaded programs on an Intel multicore machine with three types of interference. The host machine has a NUMA architecture with 16GB memory and two Intel Xeon E5620 2.40GHz 4-core processors. Each core in the processor has a private 256KB L2 cache and shares a 12MB L3 cache. Hyperthreading was disabled for all experiments. To isolate program performance from other factors, e.g., memory locality, we created the VM hosting the parallel programs in one memory node and pinned vCPUs of the VM to the processor affiliated with the node. Thus, all memory accesses were local and parallel threads shared the same last-level cache. We configured the VM running parallel programs with 4GB memory and 4 vCPUs, each pinned to a separate core in the same memory node. The background interfering VM had an identical configuration with all its vCPUs pinned to the same set of cores. Note that pinning vCPUs to pCPUs is to obtain reproducible results. We observed on average 55% performance slowdown and 15% variation when CPU affinity was turned off. We assigned equal weights to both VMs, assuming a fair allocation of the CPU.

We implemented vProfile in Xen 4.0.2 and modified Linux guest kernel 2.6.32 to use the profiling hypercalls. We selected the benchmarks in PARSEC 2.1 and the NAS Parallel Benchmark suite. The PARSEC benchmarks were compiled with *gcc-pthreads* and blocking synchronizations. We

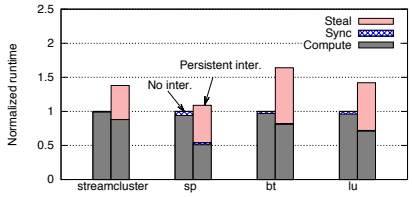

Figure 3: Interference reduces compute time.

used the OpenMP version of the NPB benchmarks and set the environment variable `OMP_WAIT_POLICY` to active to use busy-waiting synchronization. All benchmarks were configured with 4 threads. We set vProfile to report performance statistics for the entire execution after programs complete.

3.2 Varying Resilience to Interference

First, we study the slowdown at individual threads due to contentions on the CPU time. We placed a single-threaded persistent interference with one parallel thread and measured how much time was stolen (i.e., steal time) from the thread by the persistent interference. In this simple scenario, the thread co-located with the persistent loop would be the slowest thread in the parallel program, thereby deciding the overall performance. Figure 2 shows the steal time of various benchmarks relative to their compute time under the 1-loop persistent interference. Intuitively, the ratio of steal and compute time should be 1 if CPU is fairly allocated to the parallel program and the interference. However, Figure 2 suggests that some programs (i.e., *canneal*, *streamcluster* and *facesim*) be more resilient to interference and were stolen less time. An examination of program code revealed these benchmarks have fine-grained synchronizations and block frequently at synchronizing barriers. Zhou *et al.*, also showed that deliberately designed attacks can obtain excessive CPU allocations by exploiting the accounting vulnerabilities in Xen [37]. Harris *et al.* found the CPU time that each job receives can be drastically different and hard to control when multiple jobs run together [13]. In our experiments, the varying CPU allocation is due to the prioritization of latency-sensitive workloads in Xen.

Xen's credit scheduler considers vCPUs that wake up from sleep as latency-sensitive and assigns them a higher priority (i.e., the *boost* priority). Such vCPUs will preempt the current running vCPUs. Although the prioritization mechanism in theory benefits any programs that block, the granularity of synchronization plays an important role in gaining more CPU allocations. Xen implements a coarse-grained CPU scheduler which checks if the current running vCPU should be de-scheduled every 30ms (i.e., the default time slice). Whenever a vCPU is prioritized, it gains a full time slice unless voluntarily giving up the CPU, e.g., blocking due to synchronization. Thus, programs with fine-grained synchronization, e.g., those with computation less than 30ms between synchronization, are never forcibly de-scheduled by Xen due to the expiration of time slices, thereby being resilient to CPU contention. This issue is not specific to Xen and has also been observed in KVM [3].

As parallel programs have varying CPU allocations in response to interference, it is important to monitor *steal* time to determine the slowdowns at individual threads. However, the resilience to interference alone does not explain the unexpected marginal slowdown of *sp* in Figure 1, though it

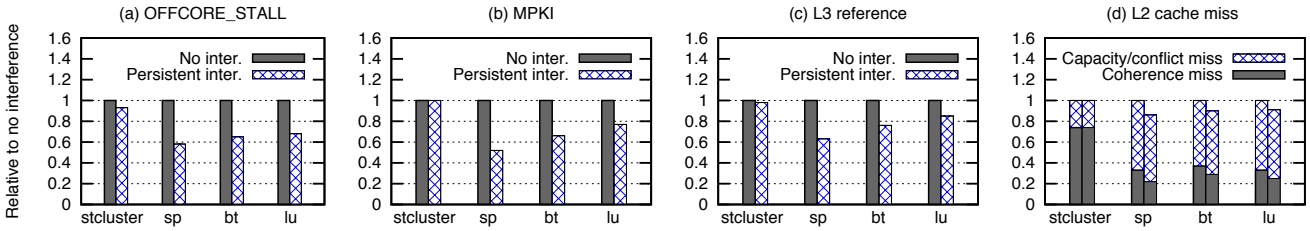

Figure 4: Interference alleviates intra-program contentions.

suffers significant steal in Figure 2. We uncover the reasons in the next subsection.

3.3 Varying Compute Time under Interference

In addition to the varying steal time, we find another factor that affects the slowdown at individual threads – the cost to access the memory hierarchy can change due to interference, leading to varying compute time. Figure 3 shows the runtime breakdown for programs that have unexpected slowdowns. These programs are expected to have slowdowns as much as their stolen time. The figure suggests that the computations needed to complete these programs decrease under the 1-loop persistent interference. The compute times for *streamcluster*, *sp*, *bt*, and *lu* in the complete program execution drop by 12%, 49%, 19%, and 29%, respectively.

To find the reasons for the reduced compute time, we show the statistics of hardware performance counters in dedicated execution and execution with 1 persistent interference in Figure 4 (a) - (d). The figures only show the statistics of the thread with interference and data is normalized to the no interference case (left bar in each group). `OFFCORE_STALL` refers to the cycles spent in the offcore memory subsystem on Intel processors. It measures the overall cost of accessing LLC and DRAM. `MPKI` measures LLC misses per thousand instructions. Figure 4 (a) suggests that these programs had fewer `OFFCORE` stalls under interference because on average there were fewer threads running simultaneously, thereby incurring less contention on the memory hierarchy. Figure 4 (b) shows that `MPKI` also drops under interference. Since the straggling thread progressed slower than sibling threads, it spent little time spinning and its instruction count decreased. A drop in the overall `MPKI` then indicates a significant drop in the number of LLC misses. Intuitively, the total LLC footprint of the parallel threads should remain unchanged, thus the LLC miss rate should not change.

We find that interference changes the way parallel applications interact with the memory hierarchy. Figure 4 (c) shows that the reduction in LLC misses was mainly due to fewer LLC references. For example, the number of LLC references for *streamcluster*, *sp*, *bt*, *lu* drop by 2%, 37%, 24%, 15%, respectively. Since LLC (L3 in our testbed) reference is the result of L2 misses, we further draw the breakdown of L2 misses in Figure 4 (d) to pinpoint the culprits of reduced L3 reference. Figure 4 (d) only shows demand data and instruction misses. Prefetching misses are excluded from the figure. We used L2 events `DEMAND.I_STATE` and `RFO.I_STATE` to count *coherence* misses and the remaining misses were *capacity* or *conflict* misses. The data shown in Figure 4 (c) and (d) is normalized to the no interference case.

From Figure 4 (d), we can see that the change in L2 coherence miss contributes most to the overall L2 miss reduction.

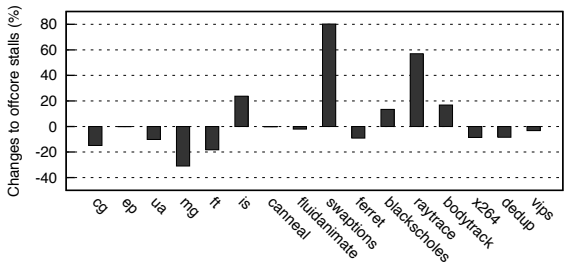

Figure 5: Interference changes memory access time.

A study of program source code reveals that most coherence misses are caused by *false sharing* between threads. The out-of-sync execution of parallel threads due to interference can constructively alleviate the contention on shared cache lines. Threads with different progression will see fewer cache line invalidations from sibling cores. Interference not only reduces the inter-core cache coherence traffic but also avoids some unnecessary L3 references. For example, miss on an unmodified L2 cacheline can be serviced by forwarding the line from a sibling core without sending requests to the L3 cache. This effectively reduces the number of capacity/conflict misses on the L3 cache. An exception is *streamcluster* whose change in compute time is mostly attributed to prefetching misses.

Next, we show that most programs in the NBP and PARSEC benchmarks have varying memory cost under interference. Figure 5 shows the changes in `OFFCORE_STALL` under interference relative to the no interference case for the benchmarks not shown in Figure 4. Negative values suggest reductions in memory cost and vice versa. As we can see, the cost to access memory varies wildly from -30.9% to 80.2%, with most programs susceptible to such changes. If the compute time comprises mostly memory access time (i.e., memory-bound), the changes in memory cost due to interference could greatly affect the overall performance. While the increase in memory access time can be attributed to the loss of locality, e.g., threads dynamically stealing work from straggling threads in *raytrace* and *bodytrack*, the reduction in memory cost due to the mitigation of intra-program contentions can be exploited to improve datacenter efficiency. We carefully co-located *sp* with four intermittent interferences, with 5%, 8%, 10%, 15% CPU demands, respectively. The setting is to create out-of-sync execution at parallel threads to mitigate the false sharing between them.

Table 2 shows that *sp* achieved 9.0% better performance with 9.2% less allocated resource. This case study **suggests** a group of symbiotic datacenter workloads, in which seemingly destructive competitions help constructively mitigate intra-program contentions on shared resources.

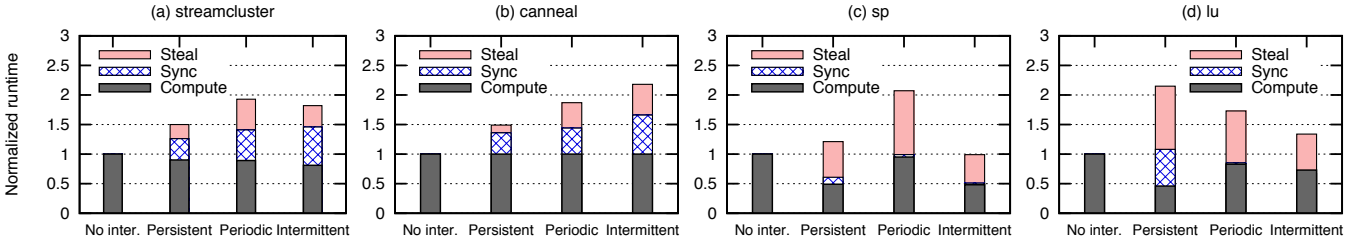

Figure 6: Program performance under different types of interference (4-loop).

	No inter.	w/ inter.
CPU %	400%	363% (**-9.2%**)
Runtime	1004s	914s (**+9.0%**)

Table 2: Improving *sp* performance with less resources.

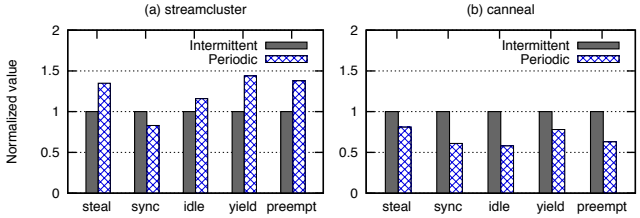

Figure 7: Low-level metrics under interference.

3.4 Complex Interactions with the Scheduler

Performance is even harder to characterize if all threads of the parallel program are affected by interference. In the simple one persistent interference scenario, program resilience to CPU contention or the change in compute time plays a major role in performance. In contrast, when all vCPUs have interference, the overall performance is a function of compute, sync, and steal time. Figure 6 shows the performance of parallel programs under different types of 4-loop interference, i.e., *persistent*, *periodic*, and *intermittent*. As shown in Figure 6(a) and (b), both *streamcluster* and *canneal* achieved the best performance under persistent interference but suffered under the other two. However, these two benchmarks behaved differently under periodic and intermittent interferences. While *streamcluster* had the largest slowdown under the periodic interference, *canneal* suffered most under the intermittent interference. Such uncertainty is due to the complex interactions among the parallel program, the interference, and the scheduler. For example, as discussed in Section 3.2, steal time is affected by the granularity of synchronization. A drop in compute time due to out-of-sync execution will decrease steal time but lead to increased sync time. The overall slowdown is determined by the interplays between these factors.

Figure 7(a) and (b) show the low-level performance metrics of *streamcluster* and *canneal* when co-running with periodic and intermittent interferences, respectively. The figures summarize the statistics for all vCPUs in the parallel VM. We find that low-level metrics shed light on the unpredictability of high-level performance. Except sync time, all other low-level metrics are highly correlated with the overall slowdown. Among these metrics, *idle* time is the key to understanding the performance difference. `Idle` refers to the time neither the parallel program nor the interference was running. A longer idle time indicates a larger overlap between parallel program's computation and the CPU burst of the interfering loop. This results in more severe contentions on the pCPU. From the figure, we can see that longer idle time always leads to more preemptions of the parallel program and more yields, which is the sign of vCPU blocking due to imbalanced execution. The **implications** from these observations are that reducing the number of preemptions would help improve performance under interference and the system idle time is a good indicator of scheduling efficiency.

For programs with busy-waiting synchronization, such as NPB benchmarks *sp* and *lu*, both spin (or sync) time and compute time are counted as the CPU consumption of the parallel VM. Since steal time depends on the CPU usage of the VM, the performance of *sp* and *lu* is determined by their combined spin and compute time. Figure 6 (c) and (d) show the runtime breakdown of *sp* and *lu*, respectively. We make three key observations. First, co-running with intermittent interference achieved the best performance among the three interference scenarios. Second, persistent interference caused longer spin time than the other two. Third, periodic interference did not significantly increase sync time. These observations provide valuable insight into the complex interplays between busy-waiting workloads and interference.

The asynchrony in scheduling multiple vCPUs contributed most to the reduction of compute time under persistent and intermittent interferences. When co-locating the parallel program with persistent interference, individual vCPUs are likely scheduled in an uncoordinated manner on multiple pCPUs and the asynchrony remains until the completion of the parallel program because the scheduling rhythm (i.e., switching vCPUs at time slice expirations) on multiple pCPUs will not change for compute-bound workloads (i.e., spinning workload and persistent interference). However, the constant asynchrony continuously incurs exceeding spinning at faster vCPUs and eventually degrades overall performance. The randomness in intermittent interference also creates asynchrony on multiple vCPUs and helps reduce memory access cost. Contrary to persistent interference, it does not cause long spin time and neither does the periodic interference. The frequent switching between computation and idling (i.e., 10ms in periodic and 1-30ms in intermittent interferences) in these two interferences forces the hypervisor to perform scheduling at a much finer granularity compared to the default 30ms time slice in Xen. There have been studies using small time slices to improve the performance of virtualized IO [1, 34]. In our case, the fine-grained scheduling helps stop spinning vCPUs in a timely manner, which not only reduces the overall sync time but also saves the precious CPU time for useful work in the parallel program. Our analysis motivates a possible **optimization** at

the hypervisor to improve the performance of busy-waiting workloads: *differentiating scheduling on multiprocessors*.

4. ONLINE PERFORMANCE PREDICTION

In this section, we present an approach for online performance prediction based on the breakdown of parallel runtime. The idea is to sample parallel execution under interference and compare the execution profile with that in an interference-free environment. The key is to compare the amount of *useful* work completed in the two profiles and infer the slowdown from the difference in the speed of program progression. *We define useful work as the necessary work needed to complete the parallel job assuming an ideal memory system with zero latency and perfect load balancing.* As we have shown, the cost of accessing memory and performing synchronization can vary under interference, leading to dynamic computations required to complete parallel programs. In a machine with an ideal memory system, computation is performed in the CPU front-end which is an invariant in the presence of interference or unpredictable thread scheduling. With perfect load balancing, the time spent in synchronization, i.e., spinning or performing context switches, is almost zero. Thus, the compute time in such an ideal platform is only determined by compilation and the dynamic instruction scheduling on individual CPUs. As these two factors are not affected by interference, the ideal compute time is a **reliable metric** to measure program progress.

To measure useful work, or the compute time on an ideal platform, we remove the time spent in the memory hierarchy, synchronization, and the time stolen by other users, from the total time. Specifically, we calculate useful work t_{ideal} in a sampling period as follows:

$$t_{ideal} = t_{total} - t_{steal} - t_{sync} - t_{mem},$$

where t_{steal} measures the resilience of the program to interference and is reported directly by vProfile. We use the time spent on Intel's offcore memory subsystem (i.e., `OFFCORE_STALL`) to approximate t_{mem}. For programs with busy-waiting synchronization, t_{sync} refers to the spinning time recorded by vProfile. For programs with blocking synchronization, t_{sync} includes the time in the blocked state and the time performing vCPU context switches. While the blocked time is already included in the execution profile, we infer the cost of context switching by comparing the cycles spent by different threads in the same parallel program. Specifically, after removing t_{steal} and t_{mem} from the total time, we compare the remaining time of individual threads and attribute the difference to the varying numbers of context switches performed by them, assuming that each thread has been assigned an equal amount of work. As such, we calculate the per context switch cost (in CPU cycles) and multiply the number of vCPU blocking to derive t_{sync}.

The online sampling profiles parallel execution in two steps. First, it enables the *dedicated mode* to collect statistics for an interference free execution. The dedicated mode sampling is then followed by a normal sampling with interference turned on. The length of the sampling can be tuned to produce the best accuracy. We empirically set the sampling length to 30 seconds. vProfile reports the runtime breakdown of all vCPUs in the VM under the two execution modes, respectively. The overall slowdown is then calculated as $\frac{t'_{ideal}}{t_{ideal}}$, where t'_{ideal} and t_{ideal} are the amount of useful work done by all threads in the parallel program during the dedicated mode and under interference, respectively.

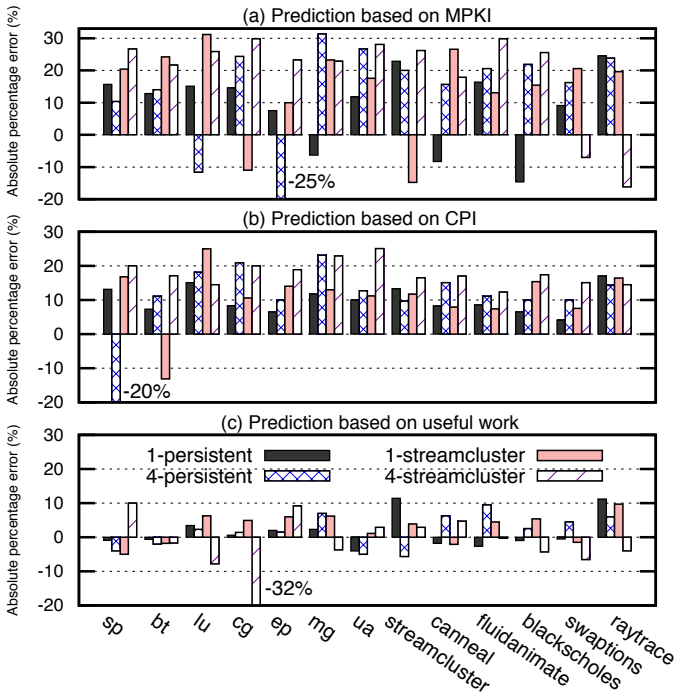

Figure 8: The accuracy of performance prediction.

Figure 8 shows the accuracy of the proposed online prediction compared with two representative prediction approaches based on misses per thousand instructions (MPKI) and cycles per instruction (CPI). We employed the same sampling-based method to predict the overall program slowdown using these two metrics. For example, the slowdown is predicted as $\frac{CPI}{CPI'}$ when using CPI to measure program progress under interference and under the dedicated mode. We used the synthetic workload and the real *streamcluster* workload as the interferences, respectively. For each type of interference, we evaluated the prediction with both the single-threaded (i.e., *1-persistent* and *1-stcluster*) and the multi-threaded (i.e., *4-persistent* and *4-stcluster*) interferences. With the synthetic persistent interference, which has predictable CPU demands and almost zero memory footprint, we test how well the prediction deals with the uncertainties due to parallel programs' varying resilience and memory accessing cost in response to interference. Then, we evaluate how accurate the prediction would be with real workload *streamcluster* that has varying CPU demands and contends on shared memory resources.

As shown in Figure 8 (a) and (b), MPKI and CPI-based predictions incurred significant prediction errors with on average 20.3% and 15.2% mean absolute percentage errors (MAPE), respectively, across all workloads. The inaccuracies were due to the misrepresentation of parallel performance by these two metrics. As shown in Figure 5, memory access cost can either increase or decrease under interference but the overall performance is determined by multiple factors. Thus, memory-related metrics alone, e.g., MPKI, fail to accurately predict performance slowdown. CPI is a effective metric to predict the performance of serial programs as it measures the cost to execute instructions, assuming that the number of instructions needed to complete a pro-

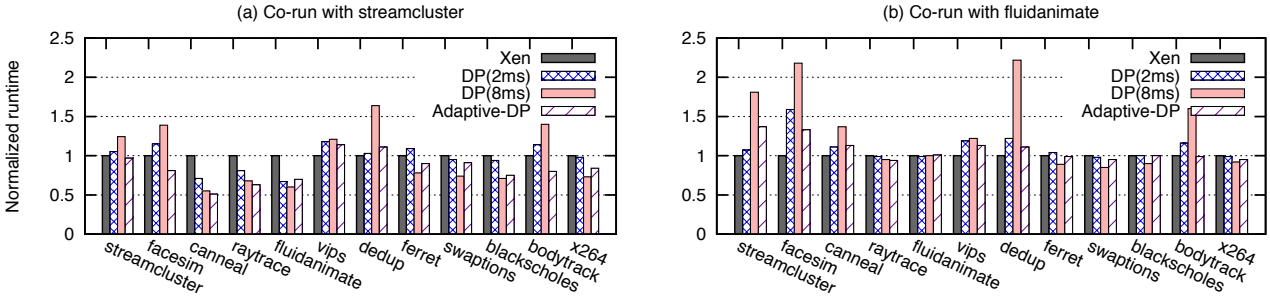

Figure 9: Performance of PARSEC benchmarks under *delayed preemption* (DP).

gram is constant. However, this assumption does not hold in parallel programs. The number of instructions executed by individual threads is variable due to the spinning and blocking performed by threads. For example, it is possible that a spinning thread making no execution progress can have a decreasing CPI because waiting on spinlocks does not cause any memory accesses and has a low CPI. Therefore, CPI alone does not capture synchronization in parallel programs and is not accurate in predicting the overall performance.

In contrast, as shown in Figure 8 (c), the performance predictions based on the useful work were accurate with an average MAPE of 4.5% across all workloads. In general, predictions with the synthetic interference are more accurate than that with *streamcluster* and predictions with single-threaded interference tend to incur less error. Except for *cg*, all predictions caused less than 10% errors even for *raytrace*, which implements dynamic work assignment at the user level to improve load balancing. While dynamic work assignment at the application level mitigates interference by assigning less work to straggling threads, it presents challenges to online performance prediction as the work done by individual threads does not reflect the overall progress. Our online prediction addressed this issue by counting useful work on all threads to measure program progress as a whole and achieved less than 10% prediction error for *raytrace*.

Sources of inaccuracy The accuracy of the prediction relies on one assumption: the sampling is representative of the overall parallel execution. This assumption do not always hold, leading to inaccurate predictions. For example, some applications, e.g., *cg*, have quite dynamic memory footprint at different execution stages. The dynamism can affect the effectiveness of the sampling-based prediction, especially when the interference is also dynamic. As shown in Figure 8, the prediction incurred 32% error on *cg* under the 4-thread *streamcluster* interference. As discussed in Section 3.2, the CPU allocation to *streamcluster* depends on its synchronization granularity. As *cg*'s memory demand changed, *streamcluster* had varying computation between synchronizations, thereby affecting the CPU allocation to *cg*. Thus, predictions based on one sample of the execution of such applications will be likely inaccurate.

5. OPTIMIZATIONS

Our analysis in Section 3 found that involuntary preemptions are especially detrimental to performance and randomness in CPU scheduling help mitigate intra-program contentions on shared resources. Inspired by these findings, we developed two simple optimizations, *delayed preemption* and *differential scheduling*, at the hypervisor to improve parallel performance under interference. Results show that the two optimizations derived from our analysis with the synthetic workloads are effective in reducing performance slowdowns when real parallel applications are co-located.

5.1 Delayed Preemption

Harmful preemptions happen when parallel programs co-locate with periodic or intermittent interference. The frequent wakeups of interfering vCPUs can cause longer steal time and synchronization time in the parallel VM. While preemptions due to the expiration of CPU time quantum are necessary for fair allocation, the boosted wakeups are needed for minimizing latencies of interactive or IO workloads. In a contended environment, such premature preemptions may cause cascading performance degradations in parallel programs. Figure 7 suggests that even CPU is heavily contended, there still exists idle time in which both parallel threads and the interfering loops are in blocked (or sleep) state. When these vCPUs wake up, preemptions between the two competing sides cause ping-pong scheduling. To address this issue, we propose *delayed preemption* (DP) to overlap computations with blocking/sleeping, and to minimize premature preemptions.

Inspired by the design of hybrid synchronization, which uses a spin-then-block approach to attain a balance between low latency and wasting CPU time, our approach temporarily delays a wakeup vCPUs for a short period of time in the hope that current running vCPU would voluntarily yield CPU due to waiting for synchronization. Our implementation in Xen is quite simple and the change only consists of 50 lines of code. We added a single shot timer to the pCPU that has a waking vCPU. If the current running vCPU voluntarily yields CPU, the timer is stopped. Otherwise, the expiration of the timer forces a call to the `schedule` function in Xen to preempt the current vCPU. However, the selection of the delayed period is challenging as different applications attain the best performance with different delays.

Figure 9 shows the performance of PARSEC benchmarks due to stock Xen and DP. We co-located PARSEC benchmarks with two background workloads. *streamcluster* has fine-grained synchronizations at the granularity of 20-30ms while *fluidanimate* has coarse-grained synchronizations every 6 seconds. Benchmarks in Figure 9(a) suffer involuntary preemptions caused by the background *streamcluster* and the background *fluidanimate* in Figure 9(b) is the victim of premature preemptions. In general, the optimal delay setting varies across benchmarks. As shown in Figure 9(a), short delay (i.e., DP(2ms)) is more desirable for *streamcluster*, *facesim*, *dedup*, and *bodytrack*, while long delay (i.e.,

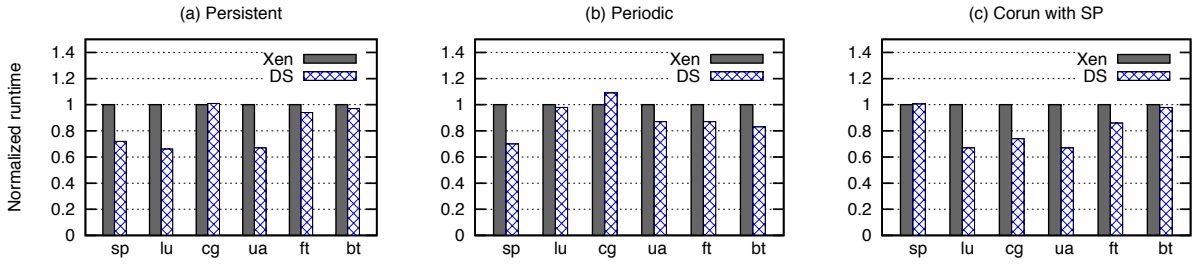

Figure 10: Performance of NPB benchmarks under *differential scheduling* (DS).

DP(8ms)) worked best for other workloads. Similarly, workloads also have their respective preferences of delay in Figure 9(b).

To meet applications' diverse needs of preemption delay, we make DP adaptive (i.e., *adaptive-DP*). As discussed in Section 3, the *idle* time on pCPUs is a key indicator of the efficiency of parallel scheduling. For PARSEC benchmarks, vCPUs become idle/blocked when synchronizing with sibling vCPUs. Interference could slowdown sibling vCPUs and prolong the idle period. In multiprocessor scheduling, uncooperative preemptions could leave CPU time on pCPUs unused when vCPUs from competing workloads are all blocked by synchronization. The objective of adaptive-DP is to dynamically adjust the preemption delays for co-running parallel workloads so that the overall preemptions and the idle time on pCPUs are minimized. We decompose runtime into *compute* and *idle* time and the latter includes *sync* and *steal* time. We adaptively change the preemption delay for a vCPU until its compute time approximates its expected fair share in a shared environment. If all vCPUs from co-running applications attain their fair share, the idle time would be dominated by the steal time, thereby minimizing the sync time and preemptions. Specifically, the preemption delay is updated as follows:

$$lag = \frac{t_{comp} - t_{fair}}{t_{comp} + t_{idle}}, \quad (1)$$

$$delay = delay + lag \times (delay + \texttt{MICROSECS(10)}), \quad (2)$$

where t_{fair} and t_{comp} refer to the ideal fair allocation of CPU time and the actual attained compute time, respectively. The preemption *delay* is updated according to the *lag* relative to the ideal fair CPU allocation. A positive lag increases the delay a vCPU waits to preempt the current running vCPU, giving other applications more time to execute. `MICROSECS(10)` is to ensure no-zero changes in case delay becomes zero.

As shown in Figure 9 (a) and (b), *adaptive-DP* automatically determines the optimal delay for different applications. When co-running with fine-grained *streamcluster* (Figure 9 (a)), adaptive-DP outperformed both DP (2ms) and DP (8ms) for applications with fine-grained synchronizations, e.g., *streamcluster*, *facesim*, and *canneal*. Adaptive-DP effectively identified better delay values for such workloads compared to the manually determined 2ms and 8ms delays. For coarse-grained workloads, such as *swaptions*, *blackscholes*, and *x264*, adaptive-DP struck a balance between the foreground and background workloads and achieved performance in-between the manually tuned delays. When co-running with *fluidanimate* (Figure 9 (b)), adaptive-DP had similar performance to stock Xen as Xen with zero preemption delays always prioritized the foreground workloads with fine-grained synchronization. We calculated the geometric mean of the slowdowns of the foreground and background workloads relative to their performance in dedicated systems. The background workloads were repeated until all foreground workload completed. Adaptive-DP outperformed Xen by 12% in the overall slowdown, indicating more efficient scheduling in a shared environment.

5.2 Differential Scheduling

Differential Scheduling (DS) is also motivated by our observations in Section 3. We found that the randomness in intermittent interference helps alleviate the contention on shared memory resources, which significantly reduces the required computation. The irregularity in CPU demand and sleep intervals forces the CPU scheduling on multiprocessors to proceed at different paces. This effectively leads to different lengths of time slice at different CPUs because the intermittent loops yield CPU at irregular intervals. The result is not only the release of pressure placed by the concurrent vCPUs to the memory hierarchy but also a reduction of wasted spinning time due to fine-grained scheduling.

To emulate the benefits brought by co-running with intermittent interference, we purposely make the schedulers on multiple CPUs have different time quantum. Xen uses a master timer for each pCPU to generate periodic timer interrupt to force a call to the `schedule` function. The default timeout (i.e., the time slice) is 30ms. In DS, the interval of the timer is randomly generated. Each time Xen sets the timeout for the next timer interrupt, it picks a random interval based on the readings of the Time Stamp Counter (tsc) register. Given the micro-second resolution (approximately 2430 cycles on our platform) of hardware timers, the last two digits of the tsc readings are likely device noises and are a good source of randomness. We set the time quantum on individual pCPUs to the range of 10ms to 30ms to enable fine-grained scheduling.

Figure 10 shows the performance of NPB benchmarks due to stock Xen and DS. Since DS already adds randomness into vCPU scheduling, we present the performance of DS when NPB benchmarks ran with persistent, periodic interferences (excluding the intermittent interference), and the real workload *sp*. The selection of *sp* is due to its memory contentiousness to co-running programs. Figure 10 (a) suggests that with persistent interference, DS was only effective in optimizing memory-bound programs, such as *sp*, *lu*, and *ua*. On average, DS outperformed Xen by 32% in these workloads. For CPU-bound program, e.g., *cg*, DS neither significantly degraded or improved performance. For periodic interference (shown in Figure 10 (b)), DS outper-

formed Xen in all benchmarks except *cg*. DS does not help mitigate memory contentions as *cg* is primarily CPU-bound and hurts performance because differential time slices slow down the tightly coupled phases in *cg*.

Interestingly, DS was able to outperform Xen even for compute-bound benchmarks, e.g., *cg* (as shown in Figure 10 (c)) when the background workload was memory-bound *sp*. The alleviation of memory pressure from the background *sp* helped improve the performance of foreground workloads. However, the inability of DS to improve foreground *sp* performance in Figure 10 (c) suggests that DS work best for workloads with complementary memory access patterns. Another advantage of DS is that the randomness in scheduling significantly reduces runtime variations across runs, with an average variation of 0.5%.

6. DISCUSSIONS AND FUTURE WORK

Enable per-thread runtime breakdown Currently, vProfile can only report runtime breakdown at per-vCPU level for parallel applications that have a one-to-one mapping from user-level threads to vCPUs. There exist many parallel workloads with more threads than vCPUs. Examples include web servers and workloads implementing worker pools, such as *dedup*, *ferret* and *vips*. Enabling per-thread runtime breakdown would help further pinpoint the source of slowdown but require tracing thread switches in the guest OS.

Extend prediction to more sophisticated workloads The online prediction currently focuses on multithreaded parallel programs on shared memory systems. Predicting parallel performance on distributed memory systems present significant challenges. Much effort is needed to devise a lightweight and accurate sampling approach on multiple machines. Further, our prediction treats the useful work on any threads equally. For more sophisticated programs, especially those having strong dependencies between threads, e.g., *dedup* and *ferret* with pipeline parallelism, the prediction should be based on the useful work of the most critical thread.

Identify symbiotic workloads Experimental results show that DP and DS are effective for blocking-based workloads (e.g., PARSEC benchmarks) and workloads with complementary memory demands, respectively. However, no one size fits all. DS can hurt the performance of CPU-bound and tightly bounded workloads, and DP is ineffective for applications with busy-waiting synchronization. Identifying symbiotic workloads that either have complementary resource demands or can be managed under similar schemes would help improve resource utilizations and reduce energy consumptions in datacenters.

7. RELATED WORK

Performance interference has been well studied in literature. Most work focused on contentions on shared resources, such as last-level caches [4, 6, 10, 12, 14], memory controllers [10, 21], and hardware prefetchers [17], between sequential programs or multi-programmed/threaded workloads. There are also recent work measuring the interference in datacenters [8, 9, 15, 19, 22, 30, 35, 36]. These studies either assume space sharing between workloads [7, 24, 27, 28], use cycles per instruction (CPI) as a proxy of performance [8, 9, 35, 36] or use offline profiling to estimate the contentiousness of co-runners [19, 22]. These approaches can not be easily extended to manage performance interference of parallel applications. First, techniques addressing single-thread resource contentions do not necessarily optimize the execution of parallel programs as parallel performance is a function of single-thread computing and synchronizations. Second, widely used metrics such as CPI are not reliable in measuring performance in parallel programs because CPI can be either inflated or deflated due to synchronizations. Third, parallel applications are usually long-running jobs. Offline profiling is prohibitively expensive in production systems. In this work, we measure the amount of useful work completed to predict slowdown in an online manner.

There are also existing studies addressing the overhead of running parallel program in virtualized environments. Our optimizations are closely related to these works. Common issues include lock-holder preemption (LHP) [32], CPU stacking [29], and expensive traps to the hypervisor [11, 26]. Co-scheduling [23, 29] aims to schedule cooperative threads synchronously to avoid the LHP issue. However, our findings in this work show that *differential scheduling* may be more desirable for memory-bound programs. *Delayed preemption* shares the similar idea with demand-based coordinated scheduling [16] to temporarily delay the preemption of important threads. The scheduler in [16] delays the preemption of vCPUs that initiate wakeup IPIs to avoid LHP. Our purpose is to interleave the computations between parallel programs and interference to avoid future harmful preemptions. Ding et. al., found that consolidating multiple threads onto one vCPU to avoid blocking leads to significant performance boost in KVM [11]. Contrary to their findings, our experiments show that programs with fine-grained synchronization are more resilient to persistent interference in a Xen environment. The contradiction can be attributed to the different designs of Linux CFS scheduler and Xen's credit scheduler.

8. CONCLUSIONS

This paper presents a systematic study of parallel performance under interference. We find that the speed of individual threads under interference is determined by their varying resilience to interferences and the computation required to complete the parallel program can change vastly under interference due to alleviated intra-program contentions. Further, the overall performance is the result of the complex interplays between these factors. Avoiding harmful vCPU preemptions or maintaining asynchrony between vCPUs helps reduce slowdown under interference for different kinds of workloads. Inspired by these findings, we develop an accurate online approach for predicting slowdowns under interference without requiring completing the parallel program, and devise two scheduling optimizations at the hypervisor to improve performance.

9. ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their constructive comments. This research was supported in part by the U.S. National Science Foundation under grants CNS-1320122, CCF-1261584 and CCF-1261811.

10. REFERENCES

[1] J. Ahn, C. H. Park, and J. Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proc of MICRO*, 2014.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Proc. of SC*, 1991.

[3] K. based virtual machine. http://www.linux-kvm.org/.

[4] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4), 2010.

[5] K. Chakraborty, P. M. Wells, and G. S. Sohi. Supporting overcommitted virtual machines through hardware spin detection. *IEEE Trans. Parallel Distrib. Syst.*, 23(2), 2012.

[6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. of HPCA*, 2005.

[7] T. Creech, A. Kotha, and R. Barua. Efficient multiprogramming for multicores with scaf. In *Proc. of MICRO-46*, 2013.

[8] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proc. of ASPLOS*, 2013.

[9] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proc. of ASPLOS*, 2014.

[10] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *Proc. of ISPASS*, 2011.

[11] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *Proc. of USENIX ATC*, 2014.

[12] F. Guo and Y. Solihin. A framework for providing quality of service in chip multi-processors. In *Proc. of MICRO*, 2007.

[13] T. Harris, M. Mass, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proc. of EuroSys*, 2014.

[14] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. Qos policy and architecture for cache/memory in cmp platforms. In *Proc. of SIGMETRICS*, 2007.

[15] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring interference between live datacenter applications. In *Proc. of SC*, 2012.

[16] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. Demand-based coordinated scheduling for smp vms. In *Proc. of ASPLOS*, 2013.

[17] F. Liu and Y. Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proc. of SIGMETRICS*, 2011.

[18] X. Liu and B. Wu. Scaanalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In *Proc. of SC*, 2015.

[19] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proc. of MICRO*, 2011.

[20] P. E. McKenney. Differential profiling. *Software - Practice and Experience*, 29(3), 1999.

[21] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. of MICRO*, 2007.

[22] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proc. of USENIX ATC*, 2013.

[23] J. Ousterhout. Scheduling techniques for concurrent systems.

[24] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A system for flexible parallel execution. In *Proc. of PLDI*, 2012.

[25] J. Rao and X. Zhou. Towards fair and efficient smp virtual machine scheduling. In *Proc. of PPoPP*, 2014.

[26] X. Song, H. Chen, and B. Zang. Characterizing the performance and scalability of many-core applications on virtualized platforms. Technical Report FDUPPITR-2010-002, Parallel Processing Institute, Fudan University, 2010.

[27] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic run-time parallelism management for time and energy efficiency. In *Proc. of ICS*, 2013.

[28] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *Proc. of PLDI*, 2014.

[29] O. Sukwong and H. S. Kim. Is co-scheduling too expensive for smp vms? In *Proc. of EuroSys*, 2011.

[30] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Proc. of ASPLOS*, 2013.

[31] The PARSEC benchmarks. http://parsec.cs.princeton.edu/.

[32] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of VM*, 2004.

[33] C. Weng, Q. Liu, L. Yu, and M. Li. Dynamic adaptive scheduling for virtual machines. In *Proc. of HPDC*, 2011.

[34] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proc. of HPDC*, 2012.

[35] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proc. of ISCA*, 2013.

[36] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proc. of EuroSys*, 2013.

[37] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and coordinated attacks in cloud computing. *J. Comput. Secur.*, 2013.