vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds

Weiwei Jia, *New Jersey Institute of Technology;* Jianchen Shan, *Hofstra University;* Tsz On Li, *University of Hong Kong;* Xiaowei Shang, *New Jersey Institute of Technology;* Heming Cui, *University of Hong Kong;* Xiaoning Ding, *New Jersey Institute of Technology*

https://www.usenix.org/conference/atc20/presentation/jia

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

vSMT-IO: Improving I/O Performance and Efficiency on SMT Processors in Virtualized Clouds

Weiwei Jia[1], *Jianchen Shan*[2], *Tsz On Li*[3], *Xiaowei Shang*[1] *Heming Cui*[3], *Xiaoning Ding*[1]
[1]*New Jersey Institute of Technology* [2]*Hofstra University* [3]*University of Hong Kong*

Abstract

The paper focuses on an under-studied yet fundamental issue on Simultaneous Multi-Threading (SMT) processors — how to schedule I/O workloads, so as to improve I/O performance and efficiency. The paper shows that existing techniques used by CPU schedulers to improve I/O performance are inefficient on SMT processors, because they incur excessive context switches and spinning when workloads are waiting for I/O events. Such inefficiency makes it difficult to achieve high CPU throughput and high I/O throughput, which are required by typical workloads in clouds with both intensive I/O operations and heavy computation.

The paper proposes to use *context retention* as a key technique to improve I/O performance and efficiency on SMT processors. Context retention uses a hardware thread to hold the context of an I/O workload waiting for I/O events, such that overhead of context switches and spinning can be eliminated, and the workload can quickly respond to I/O events. Targeting virtualized clouds and x86 systems, the paper identifies the technical issues in implementing context retention in real systems, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling.

The paper designs vSMT-IO to implement the idea and the techniques. Extensive evaluation based on the prototype implementation in KVM and diverse real-world applications, such as DBMS, web servers, AI workload, and Hadoop jobs, shows that vSMT-IO can improve I/O throughput by up to 88.3% and CPU throughput by up to 123.1%.

1 Introduction

Simultaneous Multi-Threading (SMT), or Hyper-Threading (HT) on x86 processors, is widely enabled on most cloud infrastructures [1–4]. For example, in Amazon EC2 [1], virtual instances can have their virtual CPUs (vCPUs) run on dedicated hardware threads or time-share hardware threads. With SMT, multiple hardware threads share the same set of execution resources in each core, such as functional units and caches. Thus, when enabled, SMT can effectively improve resource utilization and system throughput.

On SMT processors, CPU schedulers are critical for achieving high performance. To make optimal scheduling decisions, they must fully consider and leverage the performance features of SMT processors, particularly the intensive resource sharing between hardware threads. For example, intensive study has concentrated on symbiotic scheduling algorithms, which co-schedule the threads that can fully utilize the hardware resources with minimal conflicts on each core [5–10].

Existing scheduling optimizations for SMT processors, including symbiotic scheduling and other enhancements in existing system software [11–13], mainly target computation-intensive workloads and aim to improve processor throughput. However, the techniques that can effectively and efficiently improve the performance of I/O-intensive workloads on SMT-enabled systems have not been paid enough attention. These techniques are particularly important when a system has both computation workloads and I/O workloads, and requires both high processor throughput and high I/O throughput.

To improve I/O workload performance, existing CPU schedulers generally use two techniques, polling [14–16] and boosting the priority of I/O workloads [17–19]. However, with these techniques, I/O workloads incur high overhead on SMT processors due to busy-looping and increased context switches, which can significantly reduce the performance of computation running on other hardware threads.

This problem is particularly significant and detrimental in clouds. In clouds, I/O workloads and computation workloads are usually consolidated on the same server to improve system utilization [17, 19–22]. At the same time, virtualization is dominantly used in clouds, which causes busy-looping and context switches to incur higher overhead, because extra operations must be carried out to deschedule and reschedule virtual CPUs, as we will show in §2.

To control the overhead of polling and I/O workload priority boosting, existing system designs make trade-offs between the efficiency and the effectiveness of these techniques, which undermine the performance of I/O workloads. For polling, existing systems usually incorporate a short timeout to keep the busy-looping brief. For priority boosting, it has been a long-standing dilemma to make I/O workloads preempting the running workloads promptly or with some extra delay; to resolve this dilemma, Linux uses a scheduling delay parameter (tunable, usually a few milliseconds) as a knob to trade-off I/O workloads responsiveness and the increased context switch overhead.

Instead of improving the effectiveness-efficiency trade-off, the paper seeks a fundamental solution to the above problem. The key is a technique that can effectively improve the performance of I/O workloads with high efficiency. Our solution is motivated by the hardware-based design for efficient blocking synchronization on SMT processors [23]. With the design, blocking synchronizations can be finished efficiently without busy-looping or context switches. Specifically, the design allows a thread blocked at a synchronization point to free all its resources for other hardware threads to use, except for its hardware context; thus, when the thread is unblocked, it can resume its execution in a few cycles.

Our solution targets virtualized clouds and x86 SMT processors. It is built on a hardware-based blocking mechanism for vCPUs, named **Context Retention**. Context retention is implemented with Intel `MONITOR/MWAIT` support [24]. With context retention, when a vCPU is waiting for an I/O event, its execution context can be held on a hardware thread without busy-looping involved; upon the I/O event, the vCPU can resume execution quickly without a context switch.

1.1 Technical Issues

While the rationale of the context retention mechanism is straightforward, maximizing its potential on improving performance needs to address three technical issues listed below. These issues arise mainly because context retention may be long time periods. Many I/O operations have long latencies in millisecond scale, and the latencies may further increase due to queueing/scheduling delays. To avoid context switches, the contexts of the vCPUs waiting for the finish of these operations need to be retained on hardware threads for the same amount of time.

First, uncontrolled context retention can diminish the benefits from simultaneous multithreading, because context retention reduces the number of active hardware threads on a core. This issue is particularly serious for x86 processors, which only implement 2-way SMT[1]. When a hardware thread is used for context retention, only one hardware thread remains for computation.

Second, context retention consumes the timeslice of an I/O workload, and thus reduces its timeslice available for computation. We found that, if not well controlled, context retention can even reduce the throughput of I/O workloads.

Third, due to context retention and burstiness of I/O operations [25], the resource demand of an I/O workload may vary dramatically on a hardware thread. This makes it a challenging task to improve processor throughput with existing symbiotic scheduling methods. To determine which workloads may make fast progress if scheduled on the same core, existing symbiotic scheduling methods periodically profile workload executions and make predictions based on the profiling results. Thus, these methods are effective only when the workload on each vCPU changes steadily. They must be substantially extended to handle I/O workloads.

1.2 Major Techniques

We implement our solution and address the above issues by designing the vSMT-IO scheduling framework. It has two major components. The **Long-Term Context Retention (LTCR)** mechanism is mainly to maximize I/O throughput with high efficiency. The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to maximize processor throughput.

The LTCR mechanism mainly addresses the first two issues identified in Section 1.1. It holds the context of the vCPU waiting for an I/O event on a hardware thread for an extended time period. If the expected I/O event happens in this period, the vCPU can quickly resume and respond to the event. Otherwise, the vCPU is descheduled. The maximum length of the time period is carefully adjusted in a way that both processor throughput and I/O throughput can be improved.

With LTCR, the context of an I/O workload can be held for as long as a few milliseconds, which is more than 10x longer than the busy-looping timeout used in system software (submillisecond) [14, 15]. This makes LTCR capable of dealing with relatively high I/O latencies, which are associated with slow I/O operations (e.g., HDD accesses and SSD writes) or caused by various system factors (e.g., queueing/scheduling delay and SSD block erase). In contrast, polling is used only when I/O workloads interact with low latency devices, e.g., local network and NVMe devices [16, 26].

The RASS algorithm mainly addresses the third issue identified in Section 1.1. On each core, it classifies the vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs. It uses one hardware thread for running CPU-bound vCPUs and the other hardware thread mainly for I/O-bound vCPUs. In this way, the computation on the CPU-bound vCPUs can overlap to the greatest extent with the context retention periods on the other hardware thread. This effectively improves processor throughput, since CPU-bound vCPUs can take advantage of the hardware resources released due to context retention to make fast progress. RASS schedules CPU-bound vCPUs on both hardware threads only when I/O-bound vCPUs are not ready to run. In this case, RASS selects CPU-bound vCPUs based on the symbiosis between vCPUs (i.e., how well the vCPUs can share the hardware resources and make progress when co-scheduled).

With RASS, the first two issues identified in Section 1.1 can

[1]Though some Xeon Phi processors implement 4-way SMT, the paper targets 2-way SMT x86 processors because of their overwhelming dominance in clouds.

be further mitigated. LTCR mainly targets long context retentions. It limits the lengths of context retentions to mitigate the resource underutilization they cause and reduce the timeslice they consume. However, it cannot deal with the issues caused by relatively short context retentions. For these context retentions, RASS mitigates the resource underutilization issue (the first issue in Section 1.1) by overlapping computation and context retention; to mitigate the second issue, it helps ensure the supply of timeslice to I/O-bound vCPUs by running them on dedicated hardware threads with high priorities.

The paper makes the following contributions. First, the paper identifies the efficiency issues in existing CPU schedulers when they are used to improve I/O performance on SMT-enabled systems, and proposes a novel idea, context retention, to improve efficiency. Second, it identifies the issues in implementing the idea, and explores effective techniques to address these issues, including long term context retention and retention-aware symbiotic scheduling. Third, targeting virtualized clouds and x86 processors, the paper designs vSMT-IO to implement the idea and the techniques, and builds a system prototype based on KVM [27]. Forth, it has evaluated vSMT-IO with extensive experiments and a diverse set of 18 programs, including DBMS, web servers, AI workloads, and Hadoop jobs, and compared the performance of vSMT-IO with the vanilla system and widely-adopted enhancements. The experiments show that vSMT-IO can improve I/O throughput by up to 88.3% and processor throughput by up to 123.1%.

2 Background and Motivation

Targeting virtualized clouds, this section demonstrates the efficiency issues of existing schedulers in improving I/O performance on SMT-enabled systems. It first introduces these techniques, and experimentally verifies their inefficiency and the caused performance degradation (§2.1). Then, it explains why the issues are serious on virtualized platforms (§2.2).

2.1 Inefficient I/O-Improving Techniques

I/O-intensive applications are usually driven by I/O events. A pattern repeated in their executions is waiting for I/O events (e.g., queries received from network, or data read from disks), processing I/O events, and generating new I/O requests (e.g., responses to queries, or more disk reads). Thus, high I/O performance not only depends on fast and well-managed I/O devices to quickly respond to I/O requests. It also depends on the applications to promptly respond to various I/O events, such that new I/O requests can be generated and issued to I/O devices quickly.

Thus, CPU schedulers play an important role in improving I/O performance. To increase the responsiveness of I/O workloads to I/O events, existing schedulers use two general techniques — polling for low-latency I/O events and priority boosting for high-latency I/O events. With polling, an I/O workload waiting for an I/O event enters a busy loop (implemented with PAUSE on x86 processors) with a pre-set timeout. The workload keeps looping before it is interrupted upon the expected I/O event or is descheduled due to timeout. Thus, polling allows a workload to respond to I/O events with a minimal delay before timeouts. With priority boosting, upon an I/O event, the priority of the I/O workload is boosted, such that it can quickly preempt a running workload to respond to the I/O event.

On virtualized platforms, I/O workloads run on vCPUs; and vCPU scheduling becomes a key component affecting I/O performance. For vCPUs, polling may be implemented in guest OS kernel [28]. However, busy-looping in guest OS causes unnecessary VM_EXITs and extra overhead on x86 processors when Pause Loop Exiting (PLE) is enabled. Thus, recent designs (e.g., HALT-Polling [15]) usually implement polling at the VMM level. Priority boosting may be implemented by adjusting priorities explicitly [17] or by implicitly associating priorities with CPU time consumption. For example, Linux/KVM allows the vCPUs with lower CPU time consumption (e.g., I/O-bound vCPUs) to preempt the vCPUs with higher CPU time consumption [17,29].

Though polling and priority boosting can improve the performance of I/O workloads, they are inefficient on SMT processors. The operations associated with these techniques, busy-looping and context switches, waste the hardware resource that can be otherwise utilized by the computation on other hardware threads. Thus, the inefficiency may not be an issue when a system has only I/O workloads; but it becomes detrimental when I/O workloads are consolidated with computation workloads. Efficiency can be improved by making these techniques less aggressive, e.g., enforcing a shorter timeout for polling. However, this sacrifices the effectiveness of these techniques and I/O performance.

We illustrate the inefficiency issue with polling and priority boosting using the experiments with two combinations of applications, `Sockperf` with `Matmul`, and `Redis` with `PageRank`. `Sockperf` and `Redis` are I/O-bound. `Matmul` and `PageRank` are CPU-bound. We run each combination on a 24-core server (48 hyperthreads) with each application running in a 48-vCPU VM. This results in 2 vCPUs on each hyperthread. The VMs are managed by KVM/Linux. Detailed server/VMs configurations and application descriptions can be found in §6.

To illustrate the inefficiency issue on a well-tuned system with high efficiency, we have enhanced the HALT-Polling implementation in KVM. The enhancement makes HALT-Polling more effective, so as to further reduce context switches between vCPUs and make vCPUs more responsive to I/O events. Specifically, with the "vanilla" implementation, an idle vCPU is not allowed to perform HALT-Polling when there is another vCPU ready to run on the same hyperthread. The enhancement removes this restriction. It also increases the maximum timeout that is allowed in HALT-Polling. (HALT-Polling adjusts timeout value dynamically between 0 and

workloads	KVM w/ enhanced HALT-Polling			vSMT-IO		
	vCPU switches	spinning time	perf. imprv.	vCPU switches	spinning time	perf. imprv.
`Sockperf`	12.5K	40.1%	16.1%	3.3K	-	56.5%
`Matmul`			8.2%			57.4%
`Redis`	43.9K	27.5%	8.4%	15.1K	-	88.3%
`PageRank`			7.7%			123.1%

Table 1: *Existing techniques handling I/O workloads incur frequent vCPU switches and massive spinning, and are inefficient on SMT processors.* "vCPU switches" are counts of context switches between vCPUs every second in the server. The performance improvements are relative to "vanilla" KVM.

a maximum value.) The enhancement improves the performance of the applications by 7.7% ∼ 16.1%.

As shown in Table 1, both application combinations incur frequent vCPU switches. For example, `Redis` and `PageRank` incur a vCPU switch about every 1 millisecond on each hyperthread. At the same time, a substantial portion of CPU time is spent by polling (e.g., 40.1% for `Sockperf` and `Matmul`). vCPU switches and such massive polling inevitably degrade performance, as we will show later.

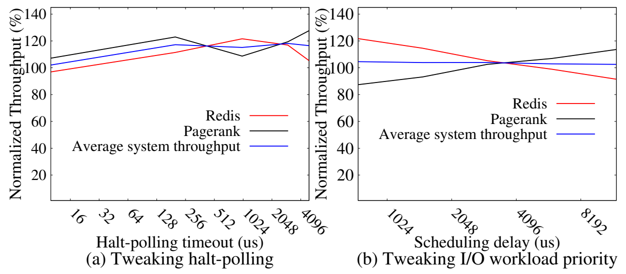

Figure 1: *Tweaking existing techniques for scheduling I/O workload cannot substantially improve performance.* (The throughputs are normalized to those with vanilla KVM.)

The performance advantage of the enhanced HALT-Polling is achieved by increasing polling to reduce costly vCPU switches. This demonstrates some potential to tweak existing designs. However, to improve performance significantly, major changes must be made. To illustrate this, Figure 1 shows how the performance of `Redis` and `PageRank` changes when tweaking the key parameters of polling and priority boosting. We first tweak the timeout used in HALT-Polling and vary it from 10 microseconds to 5 milliseconds. Figure 1(a) shows that increasing timeout only slightly improves performance when timeout value is small. However, the performance improvement of these two applications hits a plateau at about 10% after the timeout value reaches 200 microseconds.

Then, we adjust the scheduling delay parameter in Linux. The parameter controls the delay between a vCPU being woken up upon an I/O event and the vCPU preempting another vCPU. Thus, increasing the parameter essentially reduces the priority of I/O-bound vCPUs and reduces vCPU switches. As Figure 1(b) shows, the average performance barely changes; and increasing this parameter is basically sacrificing I/O performance for higher processor throughputs.

The aim of vSMT-IO is to substantially reduce the overhead caused by spinning and vCPU switches. The reduced overhead improves the performance of computation workloads. As shown in Table 1, reducing more than 2/3 of vCPU switches and eliminating spinning lead to significant performance improvement to `PageRank` (123.1% relative to vanilla KVM or 107.1% relative to enhanced KVM). More importantly, the performance improvement of computation workload is not at the cost of I/O performance. With vSMT-IO, the throughput of `Redis` is increased by 88.3% over vanilla KVM or 73.7% over enhanced KVM. The system I/O throughput is also increased by 75.1% over enhanced KVM.

2.2 Overhead of Polling and Context Switches

Existing techniques for improving I/O performance are inefficient on SMT processors, because context switches and polling waste the resource that can be otherwise utilized by the computation on other hardware threads. Targeting virtualized clouds, this subsection highlights the overhead of these operations with experiments and explains how such high overhead is incurred.

Hyperthread 1	Hyperthread 2	Relative performance
-	`Matmul`	100%
vCPUs Switches	`Matmul`	32%
HALT-Polling	`Matmul`	73%

Table 2: *vCPU switches and HALT-Polling on a hyperthread slow down the computation on the other hyperthread.*

In the experiments, we run a `Matmul` thread on a hyperthread. Then, on the other hyperthread, we make two vCPUs switch back and forth or make a vCPU repeat the HALT-Polling loop. We check how the performance of `Matmul` is impacted by these operations.

The experiments show that vCPU switches slow down `Matmul` by about 70%, and HALT-Polling slows it down by about 30% (Table 2). While the slowdowns explain the inefficiency of polling and priority boosting techniques, we were surprised at these slowdowns. We expected the slowdown caused by vCPU switches to be around 50%, because there are two streams of instructions compete for CPU resource on the hyperthreads, and expected the slowdown caused by HALT-Polling to be minimal, because PAUSE instruction is designed to consume minimal resource.

We have diagnosed the slowdowns. vCPU switches cause large slowdowns mainly because the L1 data cache shared by both hyperthreads needs to be flushed during vCPU switches to address the *L1 Terminal Fault* problem [30, 31]. Other costly operations, including TLB flush [32], handling rescheduling IPIs [33], and the execution of scheduling algorithm, also contribute to the performance impact incurred by vCPU switches. The slowdown caused by HALT-Polling is larger than expected because the operations other than PAUSE are executed. HALT-Polling is implemented in the VMM. Thus, VM_EXIT is incurred when a vCPU enters HALT-Polling. VM_EXITs are costly operations [34]. During the polling, the instructions controlling the busy-loop are executed repeatedly. They are also more costly than PAUSE.

3 Basic Idea and Technical Issues

As Section 2 shows, polling and priority boosting incur high overhead on SMT processors; tweaking these techniques yields only marginal performance improvements. This requires that a new and efficient technique be developed to handle I/O workloads.

On a SMT processor, an efficient technique must consume minimal hardware resources. In a scheduling technique for improving I/O performance, two factors determine its hardware resource consumption. One is how to handle an I/O workload while it is waiting for the completion of an I/O operation. The other is how to quickly resume the execution of the I/O workload upon the completion of the expected I/O operation. Polling and priority boosting each concentrate on reducing the resource consumption of only one factor, but at the cost of high resource consumption in the other factor. Our solution aims to minimize the resource consumption of both factors.

Our solution leverages two features of SMT processors: 1) hardware-based blocking support, and 2) intense resource sharing between hardware threads. With these features, we implement a **Context Retention** mechanism for vCPUs. While a vCPU is waiting for the completion of I/O operations, it can "block" itself on a hardware thread, and release all its resources for other hardware threads to use, except for its hardware context. This minimizes the resource consumption required by waiting for the completion of I/O operations. With the hardware context, the vCPU can be quickly "unblocked" without context switches upon the completion of the I/O operations. This minimizes the resource consumption required to quickly resume the execution of I/O workloads. Table 3 summarizes the benefits of context retention from the perspectives of both I/O workloads and computation workloads.

	Benefit	**Overhead**
I/O	better responsiveness	timeslice charged for context retention
Computation	extra resources from reduced context switches and polling	resource underutilization

Table 3: ***A summary of benefit and overhead of context retention.***

Though context retention consumes minimal hardware resources, it does incur some overhead, which are as summarized in Table 3 and must be reduced for better efficiency. From the perspective of computation workloads, because not all the hardware threads can be used for computation, the overhead is reflected by resource underutilization. Given that a x86 core has two hyperthreads, to avoid low utilization, one must be doing computation while the other does context retention. Even with this arrangement, full utilization may not be achieved.

From the perspective of I/O workloads, they are charged for vCPU usage while they retain contexts; so only short context retention periods are cheaper than descheduling and rescheduling vCPUs; but longer retention periods are not. This problem can be illustrated by the performance of I/O workload `Redis` in Figure 1(a). Increasing HALT-Polling timeout improves the performance of `Redis` when the timeout value is low. However, after the timeout exceeds 0.5 millisecond, further increasing the timeout degrades its performance. This is because, with a longer timeout, polling consumes more timeslice and reduces the timeslice available to the computation in `Redis`. Though polling is used in this experiment, if polling is replaced with context retention, the performance trend would be similar.

For the above overhead issues, a natural solution is to control the maximum length of context retention, such that extended context retention periods will not cause high overhead. However, this solution cannot deal with the overhead of the context retention periods that are relatively short. Reducing this overhead requires some enhancement in vCPU scheduling. For example, resource underutilization can be mitigated by scheduling a resource-demanding vCPU on a hyperthread when context retention is in progress on the other hyperthread; the vCPUs with much timeslice consumed by context retention can be compensated with extra timeslice.

In addition to the overhead issues, context retention also creates some challenge on the integration of symbiotic scheduling methods, which are needed for improving CPU performance. The key of symbiotic scheduling is to estimate how well a group of workloads can corun on a SMT core (i.e., symbiosis level) [6–9, 35]. This is achieved by monitoring workload executions periodically. For instance, SOS (Sample, Optimize and Symbiosis) [5] samples workload executions periodically in sample phases to determine their symbiosis levels, and preferentially coschedules tasks with the highest symbiosis levels in symbiosis phases. Thus, existing symbiotic scheduling methods require that the resource demand of a workload change steadily during its execution. Due to context retention and burstiness of I/O operations [25], the resource demand of an I/O workload changes dramatically during its execution on a vCPU. Existing symbiotic scheduling methods cannot handle such workloads. This issue may be addressed by coscheduling I/O workloads with computation workloads, such that symbiosis levels can be lifted by overlapping context retention with resource-demanding computation. Existing symbiotic scheduling methods can still be used to handle computation workloads.

4 vSMT-IO Design

We implement our idea and address the technical issues in vSMT-IO. In this section, we present the overall architecture of vSMT-IO and its major components.

4.1 Overview

Figure 2 shows the overall architecture of vSMT-IO. vSMT-IO incorporates four major components:

• The **Long Term Context Retention (LTCR)** mechanism on each core implements context retention. To prevent extended context retention periods causing high overhead (re-

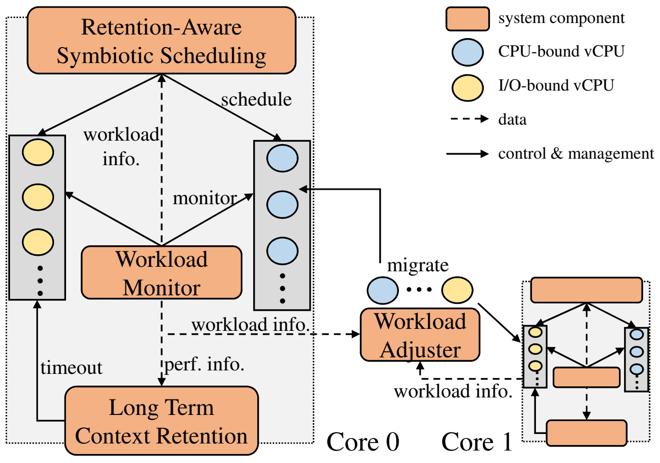

Figure 2: vSMT-IO *Architecture.* Key components are in orange.

Algorithm 1 Context Retention Timeout Adjustment

```
 1: $T_d$: desired timeout value; $T_e$: effective timeout value; $T_{init}$: initial time-
    out value; $P$: time period between two adjustments
 2: $T_d \leftarrow T_{init}$
 3: while true do
 4:     $T_e \leftarrow T_d$, collect performance data for a time period of $P$
 5:     if TESTTIMEOUT($T_d$ * 1.1) then
 6:         $T_d \leftarrow T_d * 1.1$; continue
 7:     else
 8:         $T_e \leftarrow T_d$, collect performance data for a time period of $P$
 9:     end if
10:     if TESTTIMEOUT($T_d$ * 0.9) then
11:         $T_d \leftarrow T_d * 0.9$; continue
12:     end if
13: end while
14: function TESTTIMEOUT($T$)
15:     $T_e \leftarrow T$, collect performance data for a time period of $P$
16:     $S_{cpu} \leftarrow$ average speed-up of CPU-bound vCPUs
17:     $S_{io} \leftarrow$ average speed-up of I/O bound vCPUs
18:     if $S_{cpu} > 1$ and $S_{io} > 1$ then return true; end if
19:     return false
20: end function
```

source underutilization and timeslice consumption), it enforces a context retention timeout, and dynamically adjusts the timeout value.

• The **Retention Aware Symbiotic Scheduling (RASS)** algorithm is mainly to increase the symbiosis levels of the vCPUs running on the hypertheads in each core. To achieve this, RASS classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules CPU-bound vCPUs on a hyperthread and I/O-bound vCPUs on the other hyperthread. CPU-bound vCPUs run on both hyperthreads only when I/O-bound vCPUs are not ready to run. In this way, the resource-demanding computation on CPU-bound vCPUs can overlap to the greatest extent with the resource-conserving context retention periods on I/O-bound vCPUs. Increased symbiosis levels improve CPU performance and reduce the overhead of context retentions. At the same time, using a dedicated hyperthread for I/O-bound vCPUs allows them to use extra CPU time as a "compensation" for the timeslice charged in context retention periods, and further prevents them from being unfairly penalized.

• The **Workload Monitor** on each core monitors vCPU executions. It characterizes the workloads on the vCPUs and measures performance. It provides workload information for RASS to classify and schedule vCPUs and for the workload adjuster introduced below to adjust the workloads between cores. It provides performance information for LTCR to adjust the timeout value.

• The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. The **Workload Adjuster** supplements RASS. It adjusts the workloads on each core to maintain their heterogeneity by migrating vCPUs between cores.

4.2 Long Term Context Retention (LTCR)

On x86 processors, we implement vCPU context retention with the `MONITOR/MWAIT` support. Specifically, to wait for an I/O event, a vCPU calls a `MWAIT` instruction paired with a `MONITOR` instruction that specifies a memory location in guest OS. The `MWAIT` instruction "blocks" the vCPU and keeps its context on the hyperthread. With the `MONITOR/MWAIT` support, the `MWAIT` instruction ends automatically when the content at the memory location is updated or an interrupt is directed to the hyperthread. Since both I/O events and timeouts can be notified with interrupts, we choose to use interrupts to stop `MWAIT`. To prevent `MWAIT` from being terminated by memory writes prematurely, we set the memory location used in `MONITOR` read-only.

The context retention timeout is to balance the cost and benefit of context rentention. Based on the summary in Table 3, for I/O workloads, lengthening a context retention is always a gain when it consumes less timeslice than descheduling and then rescheduling a vCPU. For computation workloads, context retention is rewarding when the amount of resource saved by reducing context switches and polling exceeds the amount of resource that cannot be utilized due to context retention. In the cases where one hyperthread does computation and the other hyperthread does context retention, context retention is always rewarding if it is not longer than the time spent on descheduling and then rescheduling a vCPU, based on the measurements shown in Table 2. Thus, context retention timeout can be set to be at least the time required by descheduling and rescheduling a vCPU. Then, longer timeouts can be tested.

LTCR uses algorithm 1 to adjust the context retention timeout periodically. The algorithm slightly increases or decreases the timeout value, checks whether performance is improved with the new value, and keeps the new value if it is. The algorithm uses the vCPU performance information collected by the workload monitor to determine whether performance is improved. Specifically, it uses IPC (instruction per cycle) to measure the performance of CPU-bound vCPUs, and uses the frequency of context retentions (i.e., number of context retentions per second) to measure the performance of I/O-bound vCPUs. Then, the algorithm calculates a speed-up for each vCPU. A speed-up value greater than 1 indicates that the performance of the vCPU has been improved with the new timeout value. It averages the speed-up values of CPU-bound vCPUs, and averages the speed-up values of I/O-bound vCPUs. The algorithm determines that the performance is improved and the new timeout value should be kept only if both average values are greater than 1.

4.3 Retention Aware Symbiotic Scheduling (RASS)

RASS schedules the vCPUs on each core with the main aim of maximizing the computation throughput of the core. This is achieved by increasing the symbiosis levels of the vCPUs running on the hypertheads. RASS combines two methods. One is *unbalanced scheduling* that maximizes the overlapping between resource-demanding computation and resource-conserving context retention periods (Section 4.3.1). The other is *symbiotic scheduling based-on cycle accounting* to select CPU-bound vCPUs with high symbiosis levels when both hardware threads need to run CPU-bound vCPUs (Section 4.3.2).

4.3.1 Unbalanced Scheduling

Unbalanced scheduling classifies vCPUs into two categories, CPU-bound vCPUs and I/O-bound vCPUs, and schedules them on paired hyperthreads (See Figure 3). The classification is based on how much time each vCPU spends on context retention. Specifically, for each vCPU, a context retention rate is calculated and updated periodically. It is the ratio between the time spent on context retention in last time period and the period length. When a new period begins, the vCPUs are ranked based on their context retention rates. The vCPUs with higher context retention rates are considered to be I/O-bound, and the rest are CPU-bound.

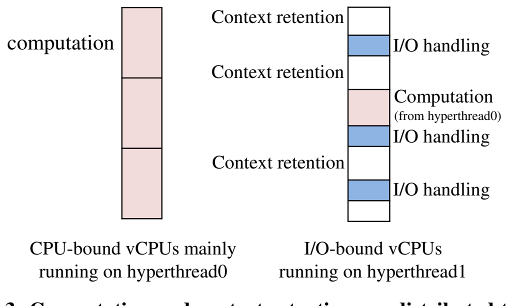

Figure 3: Computation and context retention are distributed to different hyperthreads with unbalanced scheduling.

When the hyperthread running I/O-bound vCPUs is idle, a CPU-bound vCPU is selected based on the symbiosis level (Section 4.3.2) and migrated to this hyperthread. This is to improve the utilization of CPU hardware to further increase CPU performance. The CPU-bound vCPU can only run with a priority lower than the I/O-bound vCPUs. It is preempted and migrated back when an I/O-bound vCPU becomes ready to run. This is to prevent the CPU-bound vCPU from blocking I/O-bound vCPUs and degrading I/O performance.

Unbalanced scheduling assumes that each vCPU has been attached with a weight, e.g., that used in Linux Completely Fair Scheduler (CFS). When classifying the vCPUs, it tries to balance the total weight of CPU-bound vCPUs and the total weight of I/O-bound vCPUs, and make them roughly equal. This is mainly to balance the load on the hyperthreads and reduce the migration of CPU-bound vCPUs.

The compensation to I/O-bound vCPUs for the timeslice consumed by context retentions can also be implemented by adjusting the weights of vCPUs. For example, the weights of the vCPUs can be increased based on their context retention rates. For the vCPUs that spend more time on context retentions than other vCPUs, their weights are increased by larger percentages. In this way, fewer vCPUs are classified as I/O-bound, and share the same hyperthread. However, we found that this adjustment is not necessary in most cases. The main reason is that I/O-bound vCPUs usually have low CPU utilization. Thus, even with context retention, some I/O-bound vCPUs still cannot fully consume their timeslice. Other I/O-bound vCPUs that need more timeslice acquire automatically extra timeslice as compensation. This is because the scheduler is work-conserving, and I/O-bound vCPUs have higher priority than CPU-bound vCPUs on the hyperthread and are supplied with extra timeslice first.

4.3.2 Symbiotic Scheduling Based on Cycle Accounting

When both hyperthreads need to run CPU-bound vCPUs, the symbiosis levels between vCPUs must be considered. RASS determines the symbiosis levels using the cycle accounting technique [36–39]. It is a symbiotic scheduling technique for threads. We only adapt its method that estimates the symbiosis levels between threads and use it on vCPUs.

We select this technique because of its high practicality. To estimate the symbiosis levels between threads, it samples and characterizes each individual thread, and inputs the characterization into an interference estimation model. Compared to SOS (Sample, Optimize and Symbiosis), which samples the execution of possible thread combinations [5], the cycle accounting technique has a much lower complexity ($O(n)$ vs. $O(n^2)$) and thus higher practicality.

The cycle accounting technique uses three parameters, which are the components of the CPI (cycler per instruction), to characterize a thread. The **b**ase component (**B**) is the number of cycles used to finish an instruction when all the required hardware resource and data are locally available; the **m**iss component (**M**) is the number of cycles used to handle misses (e.g., cache misses and TLB misses); the **w**aiting component (**W**) is the number of cycles waiting for hardware resource to become available. The CPI value is roughly the sum of B, M, and W.

When the parameters of a thread are being measured, the cycle accounting technique requires that the thread run alone on the core without any computation on the other hyperthread so as to eliminate interference. This incurs non-trivial overhead. To reduce this overhead, we take advantage of context retentions, and measure the parameters of a CPU-bound vCPU when it is running on a hyperthread and context retention is in progress in the other hyperthread. We obtain the base component, the miss component, and the CPI of the vCPU using hardware counters, and calculate the waiting component from this.

4.4 Workload Adjuster

The effectiveness of RASS relies on the heterogeneity of the workloads on each core, some being CPU-bound and some others being I/O-bound. Its performance advantage may diminish when workloads become homogeneous due to factors, such as load balancing and phase changes in workloads. The workload adjuster is designed to maintain the workload heterogeneity on each core.

The workload adjuster measures workload heterogeneity and characterizes the overall workload type by calculating the standard deviation and the average value of vCPU context retention rates. If a group of vCPUs have a small deviation value, their workloads are generally homogeneous; if the average context retention rate of a group of vCPUs is very high, these vCPUs are likely to be I/O-bound; if the average rate is very low, the vCPUs are likely to be CPU-bound. The workload adjuster calculates these values for each core, and updates them periodically to detect the need for workload adjustment. When the standard deviation drops below a pre-set threshold, workload adjustment starts.

To adjust the workloads, the adjuster finds the core with the smallest deviation. Then, based on the average context retention rate of the core (e.g., a very small average value of CPU-bound vCPUs), the adjuster searches for another core, which is dominated by the other type of vCPUs (e.g., I/O-bound vCPUs). The search is done by examining the average context contention rates of other cores. The desired core is the one with the average context contention rate that differs from the former average rate by the largest degree (e.g., a very large average value of I/O-bound vCPUs). After a such core is found, the adjuster ranks the vCPUs based on their context retention rates on each of these two cores, selects the vCPU ranked in the middle on each core, and swaps the two vCPUs.

5 Implementation Details

We have implemented a prototype of vSMT-IO based on Linux/KVM. We added/modified about 1300 lines of source code mainly in KVM kernel modules and Linux CFS [2]. The workload monitor and the long-term context retention (LTCR) components are mainly implemented in a KVM kernel module by changing *kvm_main.c*. In LTCR, the context retention mechanism needs to be implemented in guest OS to minimize overhead. Though it can be implemented as an idle driver kernel module [40], we choose to directly change the idle loop in *idle.c* to simplify the implementation. Context retention is implemented with a loop, which repeatedly calls `MONITOR`, `MWAIT`, and the *need_sched()* function of Linux kernel. It is inserted at the beginning of each iteration of the idle loop. Implementing context retention with a loop is to prevent it from being terminated prematurely by irrelevant interrupts. The loop terminates when a thread becomes "ready" on the vCPU (fulfilled with the *need_sched()* call). Thus, context retention can finish upon the expected I/O event. The loop also ends if a timer interrupt "marking" the timeout of the context retention is received by the vCPU. To differentiate this interrupt from regular timer interrupts, we change the two unused bits in the VM execution control register, and use them as a timeout flag.

Retention aware symbiotic scheduling and workload adjuster are implemented based on Linux CFS in *fair.c* and *core.c*. Thus, the original scheduling and load balancing policies implemented in CFS are followed in most cases, e.g., when deciding which I/O-bound vCPU is the next to run on a hyperthread. However, when deciding which CPU-bound vCPU is the next to run, the symbiotic scheduling policy in RASS and the fairness based scheduling policy in CFS have different objectives, and thus may decide to select different vCPUs. To coordinate these different objectives, our implementation let Linux CFS select a few vCPUs based on its policies. Then, among these vCPUs, RASS selects a vCPU based on symbiosis.

6 Performance Evaluation

With the prototype implementation, we have evaluated vSMT-IO extensively with a diverse set of workloads. The objectives of the evaluation are four-fold: 1) to show that vSMT-IO can improve I/O performance with high efficiency and benefit both I/O workload and computation workload, 2) to verify the effectiveness of the major techniques used in vSMT-IO, 3) to understand the performance advantages of vSMT-IO across diverse workload mixtures and different scenarios, and 4) to evaluate the overhead of vSMT-IO.

6.1 Experiment Settings

Our evaluation was done on a DELL™ PowerEdge™ R430 server with two 2.60GHz Intel Xeon E5-2690 processors (two NUMA zones), 64GB of DRAM, a 1TB HDD, and an Intel I350 Gigabit NIC. Each processor has 12 physical cores, and each physical core has two hyperthreads. With KVM, we built four VMs, each with 24 vCPUs and 16GB memory. Both the host OS and guest OS are Ubuntu Linux 18.04 with kernel updated to 5.3.1. We test vSMT-IO with a large and diverse set of workloads generated by typical applications from different domains, as summarized in Table 4. In the experiments, each VM encapsulates one workload.

We test vSMT-IO under two settings. Under the first setting, we launch two VMs; thus each vCPU has a dedicated

[2]Source code can be found at https://github.com/vSMT-IO/vSMT-IO.

App.	Workload Description
Redis	Serve requests (randomly chosen keys, 50% SET, 50% GET) [42].
HDFS	Read 10GB data sequentially with HDFS TestDFSIO [43].
Hadoop	TeraSort with Hadoop [43].
HBase	Read and update records sequentially with YCSB [44].
MySQL	OLTP workload generated by SysBench for MySQL [45].
Nginx	Serve web requests generated by ApacheBench [46].
ClamAV	Virus scan a large file set with clamscan [47].
RocksDB	Serve requests (randomly chosen keys, 50% SET, 50% GET) [48].
PgSQL	TPC-B-like workload generated by PgBench [49].
Spark	PageRank and Kmeans algorithms in Spark [50].
DBT1	TPC-W-like workload [51].
XGBoost	Four AI algorithms included in XGBoost [52] system.
Matmul	Multiply two 8000x8000 matrices of integers.
Sockperf	TCP ping-pong test with Sockperf [53].

Table 4: Benchmark applications used to test vSMT-IO.

hyperthread. We compare vSMT-IO against three competing solutions: 1) Blocking, which immediately deschedules the vCPUs waiting for I/O events, and is implemented by disabling HALT-Polling in KVM; 2) Polling, which is implemented by booting guest OS with parameter *idle=poll* configured [41] (timeout is not enforced for best I/O performance); and 3) HALT-Polling implemented in KVM, which combines polling and priority boosting techniques.

Under the second setting, we launch four VMs; thus, each hyperthread is time-shared by two vCPUs. Without a timeout, Polling is not a choice for improving I/O performance under this setting. Thus, we compare vSMT-IO against 1) vanilla KVM, which uses priority boosting to improve I/O performance, because HALT-Polling implemented in vanilla KVM is inactive under this setting, and 2) HALT-Polling enhanced to support time-sharing (described in Section 2.1).

We measure the throughputs of the workloads. We also collect response times if the workloads report them. The performance measurements may vary significantly across different workloads. When we present them in figures, for clarity, we normalize them against those of Blocking under the first setting and priority boosting (i.e., vanilla KVM) under the second setting.

6.2 One vCPU on Each Hyperthread

Under the first setting, I/O workloads can achieve the best performance with Polling. We want to compare the effectivenss of vSMT-IO on improving I/O performance against that of Polling by comparing the performance of I/O workloads managed with these two solutions. Without a timeout, Polling incurs high overhead on SMT processors, and degrades the performance of other workloads on the processors. Blocking and HALT-Polling are more efficient solutions than Polling under this setting. We want to compare the efficiency of vSMT-IO against that of Blocking and HALT-Polling by comparing the performance of computation workloads when they are collocated with I/O workloads managed with these three solutions.

With the above objectives, we launch two VMs. We run Matmul in one VM, which is computation-intensive, and run an I/O-intensive benchmark in the other VM. Figure 4 shows the normalized throughputs of Matmul and eight I/O-intensive benchmarks selected to co-run with Matmul. Note that the performance with Blocking is shown with the flat line at 100%.

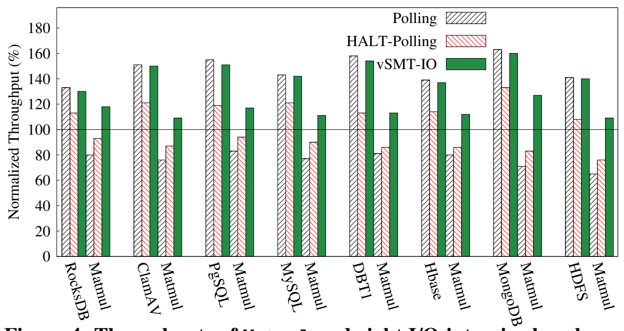

Figure 4: Throughputs of Matmul and eight I/O-intensive benchmarks when Matmul is collocated with each of the benchmarks in two VMs. Each vCPU runs on a dedicated hyperthread. Throughputs are normalized to those of Blocking.

With vSMT-IO, the I/O-intensive benchmarks achieve similar performance as they do with Polling. The largest difference is with DBT1, 4.1%. This is because DBT1 incurs a large number of random accesses to the HDD, which have long latencies exceeding the timeout value used in LTCR. On average, the I/O intensive benchmarks are only 2.3% slower with vSMT-IO. This shows that vSMT-IO is highly effective on improving I/O performance.

The high effectiveness of vSMT-IO is achieved with high efficiency. This is reflected by Matmul achieving higher performance with vSMT-IO consistently in all the experiments than it with the other three solutions. On average, with vSMT-IO the performance of Matmul is 37.9%, 14.5%, and 27.6% higher than it with Polling, Blocking, and HALT-Polling, respectively.

6.3 Multiple vCPUs Time-Sharing a Hyperthread

With multiple vCPUs on each hyperthread, context switches are usually incurred when improving I/O performance. It becomes more difficult for I/O-improving solutions to maintain high efficiency. We want to know to what extent the effectiveness and efficiency of vSMT-IO can be maintained. At the same time, vSMT-IO can be fully exercised under this setting. We want to verify the effectiveness of the major techniques in vSMT-IO.

In the experiments, we launch four VMs. On two of the VMs, we run two instances of the same benchmark, which is computation-intensive, e.g., Nginx, or AI algorithms in XGBoost. On the other two VMs, we run two instances of another benchmark, which is I/O-intensive, e.g., web server, or file server.

Figure 5 shows the normalized throughputs for eight benchmark pairs. In each pair, the first benchmark is I/O intensive, and the second benchmark is computation intensive. The enhanced HALT-Polling can effectively improve the throughputs of I/O-intensive benchmarks, because polling can "absorb" some context switches caused by I/O operations. Compared to vanilla KVM, the throughputs of I/O intensive benchmarks are increased by 36.9% on average. However, polling consumes CPU resources and may degrade the performance of other workloads (e.g., Nginx and Regression). Because the length of polling is carefully controlled in HALT-Polling, on average the throughputs of computation-intensive benchmarks are similar to those with vanilla KVM.

Compared to enhanced HALT-Polling, vSMT-IO can more effectively improve the throughputs of I/O-intensive benchmarks. On average, their throughputs are 29.5% higher than those with enhanced HALT-Polling. More importantly, this is achieved by improving the throughputs of computation-intensive workloads at the same time. On average, the throughputs of computation-intensive workloads with vSMT-IO are 22.8% and 18.4% higher than those with enhanced HALT-Polling and vanilla KVM, respectively.

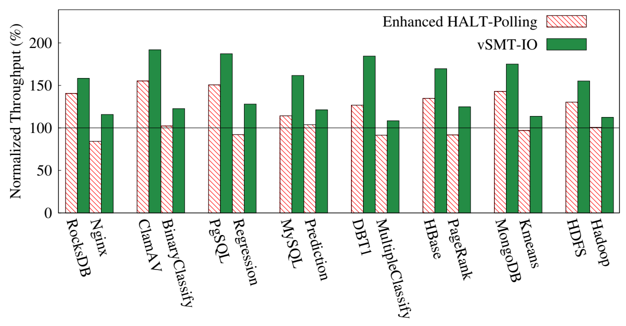

Figure 5: Throughputs of eight pairs of benchmarks. Each benchmark has two instances running on two VMs. Each hyperthread is time-shared by 2 vCPUs. Throughputs are normalized to those with vanilla KVM. Benchmarks BinaryClassify, MultipleClassify, Regression and Prediction are AI algorithms in XGBoost [52, 54].

The results in Figure 5 confirm that vSMT-IO can maintain its effectiveness and efficiency when each hyperthread is time-shared by vCPUs. To further investigate how the throughputs are improved with vSMT-IO, we collect the frequencies of vCPU switches (shown in Table 5) and profile the workload on the hyperthreads for I/O-bound vCPUs (results shown in Table 6).

The effectiveness of vSMT-IO on improving I/O performance relies on context retentions holding vCPU contexts on hyperthreads (the LTCR component). It is reflected by reduced context switches. As shown in Table 5, vSMT-IO can reduce vCPU switches significantly by up to 95% (80% on average). As a comparison, enhanced HALT-Polling can only reduce vCPU switches by at most 51% (32% on average). This explains the superiority of vSMT-IO over HALT-Polling.

The high efficiency of vSMT-IO comes partially from its capability to reduce vCPU switches. It also comes from LTCR and RASS controlling the overhead incurred by context

Benchmark Pairs	Number of vCPU Switches Per Second		
	Vallina KVM	Enhanced HALT-Polling	vSMT-IO
(RocksDB,Nginx)	29.3k	15.2k	1.9k
(ClamAV,BinaryClassify)	11.8k	8.7k	3.2k
(PgSQL,Regression)	9.5k	8.0k	2.8k
(MySQL,Prediction)	11.5k	9.3k	4.5k
(DBT1,MultipleClassify)	61.3k	29.5k	3.9k
(HBase,PageRank)	23.4k	12.3k	3.9k
(MongoDB,Kmeans)	33.3k	20.8k	9.3k
(HDFS,Hadoop)	34.0k	30.6k	1.7k

Table 5: The number of vCPU switches is substantially reduced with vSMT-IO for the eight benchmark pairs.

Benchmark Pairs	Context Retentions	I/O Workload	Computation Workload
(RocksDB,Nginx)	28.1%	34.3%	37.6%
(ClamAV,BinaryClassify)	39.8%	31.6%	28.6%
(PgSQL,Regression)	42.3%	19.2%	38.5%
(MySQL,Prediction)	30.0%	33.5%	36.5%
(DBT1,MultipleClassify)	32.7%	54.4%	12.9%
(HBase,PageRank)	53.9%	31.9%	14.2%
(MongoDB,Kmeans)	34.4%	45.3%	20.3%
(HDFS,Hadoop)	33.0%	45.2%	21.8%

Table 6: Time (percentage) spent by context retentions, I/O-bound vCPU, and CPU-bound vCPU on the hyperthreads for I/O-bound vCPUs.

retentions. While the effectiveness of RASS on controlling the overhead is self-evident, the effectiveness of LTCR can be confirmed with the results shown in Table 6. LTCR limits the context retention lengths to prevent high overhead. As a result, on the hyperthreads for I/O-bound vCPUs, for most benchmark pairs, the time spent on context retentions is less than 40%. With context retention lengths well controlled, more than 20% of the CPU time on these hyperthreads can be used by CPU-bound vCPUs to improve CPU throughput.

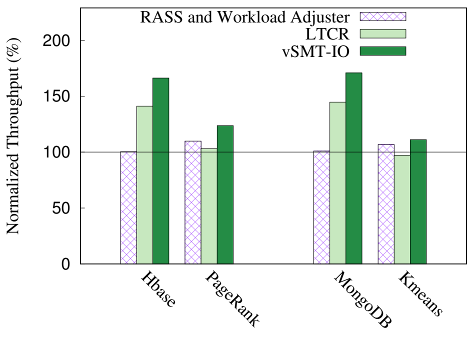

Figure 6: Normalized throughputs (relative to those achieved with vanilla KVM) of two pairs of benchmarks when LTCR and RASS are enabled separately.

To understand how the two major techniques in vSMT-IO, LTCR and RASS, improve performance, we enable these techniques separately, and show the performance of two pairs of benchmarks, HBase with PageRank, and MongoDB with

`Kmeans`, in Figure 6. `Workload Adjuster` is enabled along with `RASS`, because it is a supplement to `RASS`. Figure 6 shows that the performance improvements of I/O-intensive workloads are mainly from the `LTCR` technique; and the performance improvements of computation-intensive workloads are mainly from the `RASS` technique. When `LTCR` is enabled, the throughputs of I/O-intensive workloads, `HBase` and `MongoDB`, are significantly increased by 41.1% and 44.7%, respectively. However, it barely increases the throughputs of `PageRank` and `Kmeans`. Further enabling `RASS` (with `Workload Adjuster`) can effectively improve the throughputs of all the workloads.

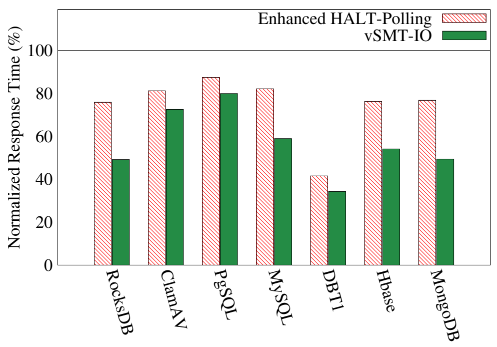

Figure 7: Response times of `RocksDB`, `ClamAV`, `PgSQL`, `MySQL`, `DBT1`, `HBase`, and `MongoDB` normalized to those with vanilla KVM (shown with the horizontal line at 100%).

Some benchmarks report response times. Figure 7 compares how their response times are reduced with vSMT-IO and `HALT-Polling`. Relative to vanilla KVM, `HALT-Polling` reduces the response times by 28.7% on average. vSMT-IO can reduce the response times by larger percentages (50.8% on average). To investigate how vSMT-IO reduces response times, we monitor the state changes of the vCPUs during the executions of these benchmarks, collect the time spent by vCPUs at the following states: 1) ***Running***, including context retention, on a hyperthread, 2) ***Ready*** and waiting to be scheduled, 2) ***Waiting*** for an event. In Table 7, for each benchmark, we show the time (in milliseconds) spent in these states for serving a request.

Benchmark	Vallina KVM			Enhanced `HALT-Polling`			vSMT-IO		
	Run	Ready	Wait	Run	Ready	Wait	Run	Ready	Wait
`RocksDB`	116.2	132.6	378.1	131.7	88.0	305.4	129.8	69.0	237.2
`ClamAV`	15.2	45.7	10.9	12.9	29.7	10.5	11.0	21.1	9.5
`PgSQL`	14.7	37.6	10.1	12.3	27.1	9.4	13.5	19.7	8.7
`MySQL`	111.4	319.7	88.9	90.7	167.9	89.5	87.5	136.7	80.6
`DBT1`	346.2	1831.4	1390.2	361.6	842.9	1035.8	306.9	643.2	641.6
`HBase`	266.2	655.0	901.8	237.8	323.3	795.9	241.6	315.0	654.3
`MongoDB`	376.1	528.6	1444.1	365.3	345.2	1276.6	351.7	256.0	897.9

Table 7: Time spent by vCPUs in three states when processing a request with vanilla KVM, enhanced `HALT-Polling`, and vSMT-IO.

The response times are reduced with vSMT-IO mainly because vCPUs spend less time on waiting to be scheduled or for events. As shown in Table 7, vSMT-IO can significantly reduce the time in the *Ready* state (53.6% on average). This is because context retention reduces context switches between vCPUs, and thus reduces the scheduling delay associated with the switches. We have noticed that the time in the *Waiting* state is substantially reduced for some benchmarks (e.g., `DBT1`). This is because finising an I/O operation sometimes need the collaboration of multiple vCPUs in the VM. For example, after a vCPU sends out an I/O request and becomes idle, another vCPU may receive the response and must notify the former vCPU by sending it an inter-processor interrupt (IPI). In this case, reducing the *Ready* time of the latter vCPU (i.e., scheduling it earlier) can also reduce the *Waiting* time of the former vCPU.

6.4 Applicability and Overhead

vSMT-IO targets heterogeneous workloads with intensive I/O operations and heavy computation. We want to know how well vSMT-IO performs for the workloads with different heterogeneity. This subsection tests the performance and overhead of vSMT-IO for different workload mixes. We still use 4 VMs to run 4 instances of 2 applications in the experiments. But we change VM sizes (i.e., the number of vCPUs in a VM) to change the workload mix. For example, to make the workload more I/O-intensive, we increase the sizes of the 2 VMs running I/O-intensive benchmarks and reduce the sizes of the VMs running computation-intensive benchmarks. The total number of vCPUs of the 4 VMs is kept fixed (96 vCPUs).

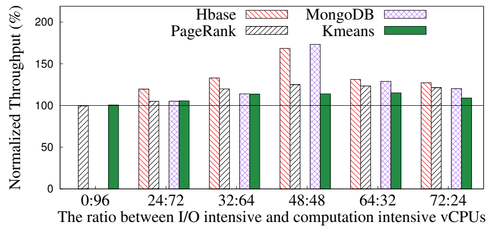

Figure 8: Normalized throughputs of vSMT-IO under different workload mixes. Throughputs are normalized to those with vanilla KVM.

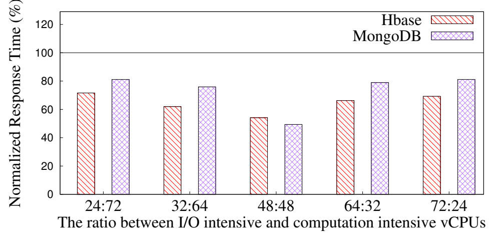

Figure 9: Normalized response times of vSMT-IO under different workload mixes. Response times are normalized to those with vanilla KVM.

Figure 8 shows the normalized throughputs of two benchmark paris, `HBase` with `PageRank`, and `MongoDB` with `Kmeans`, when the VM sizes for I/O-intensive benchmarks and computation-intensive benchmarks are changed from (12,36) to (36,12). (The ratios of the vCPUs running these benchmarks vary from 24:72 to 72:24.) Figure 9 shows the response times of `HBase` and `MongoDB` in these experiments. Though vSMT-IO can improve performance for all these workload mixes, it improves performance by the largest percentages when the number of vCPUs running I/O-intensive benchmarks is the same as the number of vCPUs running computation-intensive vCPUs.

We also run `PageRank` and `Kmeans` in two VMs with 48 vCPUs each, and show the normalized throughputs (labeled with "0:96") in Figure 8. Because both benchmarks are computation intensive, there is no space for vSMT-IO to improve performance. The performance difference between vSMT-IO and vanilla KVM is unnoticeable (less than 2%). This shows that the overhead of vSMT-IO is very low.

We have also evaluated the performance of vSMT-IO with 8 VMs (192 vCPUs). We find that vSMT-IO consistently shows better performance than vanilla KVM and enhanced `HALT-Polling`, for heterogeneous workloads; but the performance improvement is similar to that with 4 VMs. The performance advantage of vSMT-IO is more determined by the mix of workloads than the number of VMs on each server.

7 Related Work

Improving I/O performance in virtualized systems. I/O performance problems in virtualized systems have been intensively studied; and various solutions have been proposed, including shortening time slices [55–58], task-aware priority boosting [17, 18, 59–69], and task consolidation [19, 70–73]. These solutions are not designed for SMT processors, and are orthogonal to our work. Shortening time slices of vCPUs can reduce the latency of I/O workloads in virtualized systems. However, it may incur significant performance degradation caused by context switches. Task-aware priority boosting improves I/O performance in virtualized systems by prioritizing I/O-intensive workloads. For instance, xBalloon [17] maintains the high priority of I/O-intensive workloads by reserving CPU resource for them. However, this may hurt the performance of computation-intensive workloads. vMigrater [19] prioritizes I/O-intensive workloads by migrating them away from to-be-descheduled vCPUs to other vCPUs, such that they can keep running and generating I/O requests. However, it is designed for VMs with multiple vCPUs, and may incur high workload migration cost. Task consolidation solutions can improve I/O performance by reducing the descheduling and rescheduling of vCPUs. They consolidate workloads onto fewer vCPUs if the workloads are I/O-intensive, such that these vCPUs can be kept active with relatively low cost. These solutions may also incur high cost due to frequent workload migrations. Polling is used in these solutions to keep vCPUs active. This is inefficient on SMT processors and can be improved by replacing polling with context retention.

Symbiotic scheduling aims to maximize the throughput of SMT processors by selecting the tasks with complementary resource demands and coscheduling them on the same SMT core [5–10]. For instance, SOS (Sample, Optimize, Symbiosis) and its variants [5–10, 35] sample task executions when they are coscheduled onto the same core, and preferentially coschedule those with small slowdowns. These solutions only target processor throughput, and cannot be used to improve the performance of I/O-intensive workloads.

Other scheduling solutions for SMT processors. Instead of maximizing processor throughput, some scheduling solutions aim to secure resources for individual tasks on SMT processors to ensure their decent performance [11, 35, 74, 75]. For instance, ELFEN [11] aims to ensure the high performance of latency-critical tasks when they are collocated with batch tasks on SMT processors. It puts a latency-critical task and batch tasks on different hardware threads in the same core, and "blocks" batch tasks when the latency-ciritical task is making progress. The efficiency is low with this solution, because each core has only one active hardware thread at any moment, and resource is underutilized. Tasks on the same SMT processor may not share the resources in a fair way. Various solutions have been proposed to enforce fairness among the tasks in a SMT-enabled system [76–78]. For instance, progress-aware scheduler [76] periodically estimates the progress of tasks, and prioritizes the tasks with relatively slow progress. vSMT-IO is orthogonal to these solutions. It increases efficiency to improve both CPU performance and I/O performance.

8 Conclusion and Future Work

Despite the prevalence of SMT processors, the problems with how to improve I/O performance and efficiency on SMT processors are surprisingly under-studied. Existing techniques used in CPU schedulers to improve I/O performance are seriously inefficient on SMT processors, making it difficult to achieve high CPU throughput and high I/O throughput. Leveraging the hardware feature of SMT processors, the paper designs vSMT-IO as an effective solution. The key technique in vSMT-IO is context retention. vSMT-IO targets virtualized clouds and x86 systems and addresses a few challenges in implementing context retention in real systems. Extensive experiments confirm its effectiveness.

NUMA systems have become ubiquitous. Though our evaluation demonstrates that vSMT-IO achieves better performance than competing solutions, the designs in vSMT-IO have not been optimized for NUMA systems. As future work, we want to make vSMT-IO "NUMA-aware" to further improve its performance. For example, the workload adjuster can be enhanced by adjusting workloads within each NUMA node before it migrates vCPUs across NUMA nodes.

9 Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Dr. James Bottomley for his helpful suggestions as the shepherd for this paper. This work is funded in part by the US National Science Foundation under a grant CCF 1617749, HK RGC ECS (No.27200916), HK RGC GRF (No.17207117 and No. 17202318), and a Croucher Innovation Award.

References

[1] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/#instance-details.

[2] SMT Configurations in VMWARE. https://bit.ly/1LxQTiW.

[3] Introducing hyperthreading into azure vms. https://azure.microsoft.com/en-us/blog/introducing-the-new-dv3-and-ev3-vm-sizes/.

[4] Google Cloud Virtual Machine Types. https://cloud.google.com/compute/docs/machine-types.

[5] Allan Snavely and Dean M Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *ACM SIGPLAN Notices*, 35(11):234–244, 2000.

[6] Jun Nakajima and Venkatesh Pallipadi. Enhancements for hyper-threading technology in the operating system: Seeking the optimal scheduling. In *WIESS*, pages 25–38, 2002.

[7] Kefeng Deng, Kaijun Ren, and Junqiang Song. Symbiotic scheduling for virtual machines on SMT processors. In *2012 Second International Conference on Cloud and Green Computing*, pages 145–152. IEEE, 2012.

[8] James R Bulpin and Ian Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*, pages 399–402, 2005.

[9] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. L1-bandwidth aware thread allocation in multicore SMT processors. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 123–132. IEEE Press, 2013.

[10] Alexandra Fedorova, Margo Seltzer, and Michael D Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, volume 33, pages 10–17, 2006.

[11] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX Annual Technical Conference*, pages 309–322, 2016.

[12] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip multi processing aware linux kernel scheduler. In *Linux Symposium*, page 193. Citeseer, 2005.

[13] Matt Liebowitz, Christopher Kusek, and Rynardt Spies. *VMware VSphere performance: designing CPU, memory, storage, and networking for performance-intensive workloads*. John Wiley & Sons, 2014.

[14] KVM: Dynamic Halt Polling Patches. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=aca6ff29c4063a8d467cdee241e6b3bf7dc4a171.

[15] Dynamic Halt Polling Technique. https://lkml.org/lkml/2017/6/22/296.

[16] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[17] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 269–281. ACM, 2017.

[18] Luwei Cheng and Cho-Li Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 2. ACM, 2012.

[19] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. Effectively mitigating i/o inactivity in vcpu scheduling. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, 2018. USENIX Association.

[20] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 329–341, Lombard, IL, 2013. USENIX.

[21] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[22] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[23] Dean M Tullsen, Jack L Lo, Susan J Eggers, and Henry M Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58. IEEE, 1999.

[24] Intel 64 and ia-32 architectures developer's manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[25] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Winter USENIX Conference*, page 405–420, 1993.

[26] Damien Le Moal. I/o latency optimization with polling. *Vault–Linux Storage and Filesystem*, 2017.

[27] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[28] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding advanced storage controller functionality via low-overhead virtualization. In *FAST*, volume 12, pages 15–15, 2012.

[29] Chandandeep Singh Pabla. Completely fair scheduler. *Linux J.*, 2009(184), August 2009.

[30] L1 Terminal Fault. https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[31] Flushing L1 Data Cache When a vCPU Enters the Guest OS. https://lore.kernel.org/patchwork/patch/974356/.

[32] Flushing TLB When a vCPU Enters the Guest OS. https://lwn.net/Articles/740363/.

[33] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *ACM SIGPLAN Notices*, volume 48, pages 369–380. ACM, 2013.

[34] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using SMT to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 750–761. ACM, 2019.

[35] Allan Snavely, Dean M Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30, pages 66–76. ACM, 2002.

[36] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A performance counter architecture for computing accurate cpi components. *ACM SIGOPS Operating Systems Review*, 40(5):175–184, 2006.

[37] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in SMT processors. *ACM Sigplan Notices*, 44(3):133–144, 2009.

[38] Josue Feliu, Stijn Eyerman, Julio Sahuquillo, and Salvador Petit. Symbiotic job scheduling on the ibm power8. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 669–680. IEEE, 2016.

[39] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 91–102. ACM, 2010.

[40] The HALT-Polling Kernel Module. `https://patchwork.kernel.org/patch/11030651/`, 2019.

[41] The Halt Polling Technique. `https://lwn.net/Articles/384146/`.

[42] Redis In-memory Key-Value Database. `http://redis.io/`.

[43] Apache Hadoop Systems. `http://hadoop.apache.org/core/`.

[44] Yahoo! Cloud Serving Benchmark. `https://github.com/brianfrankcooper/YCSB`, 2004.

[45] MySQL Database. `http://www.mysql.com/`, 2014.

[46] Apache Web Server. `http://www.apache.org`, 2012.

[47] Clam AntiVirus Benchmarks. `http://www.clamav.net/`.

[48] RocksDB NoSQL Storage System. `https://rocksdb.org/`.

[49] PostgreSQL DBMS Benchmarks. `https://www.postgresql.org`, 2012.

[50] Apache spark benchmarks. `https://spark.apache.org/examples.html`.

[51] TPC-W Database Benchmarks. `http://osdldbt.sourceforge.net/`.

[52] XGBOOST Runtime System. `http://dmlc.cs.washington.edu/xgboost.html`.

[53] Sockperf. `https://github.com/Mellanox/sockperf`.

[54] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[55] Cong Xu, Sahan Gamage, Pawan N Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. vslicer: latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[56] Jeongseob Ahn, Chang Hyun Park, and Jaehyuk Huh. Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 394–405. IEEE Computer Society, 2014.

[57] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. Accelerating critical os services in virtualized systems with flexible micro-sliced cores. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 29:1–29:14, New York, NY, USA, 2018. ACM.

[58] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 3:1–3:14, 2016.

[59] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 101–110. ACM, 2009.

[60] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[61] Xiaoning Ding, Phillip B Gibbons, and Michael A Kozuch. A hidden cost of virtualization when scaling multicore applications. In *HotCloud*, 2013.

[62] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[63] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vread: Efficient data access for hadoop in virtualized clouds. In *Proceedings of the 16th Annual Middleware Conference*, pages 125–136. ACM, 2015.

[64] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[65] Hui Lu, Cong Xu, Cheng Cheng, Ramana Kompella, and Dongyan Xu. vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 453–460. IEEE, 2015.

[66] Sahan Gamage, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Opportunistic flooding to improve tcp transmit performance in virtualized clouds. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 24. ACM, 2011.

[67] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 125–138. ACM, 2015.

[68] Ron C Chiang and H Howie Huang. Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.

[69] Weiwei Jia Jianchen Shan and Xiaoning Ding. Rethinking the scalability of multicore applications on big virtual machines. In *IEEE International Conference on Parallel and Distributed Systems*. IEEE, 2017.

[70] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule processes, not VCPUs. In *APSys 2013*, pages 1:1–1:7, 2013.

[71] Luwei Cheng, Jia Rao, and Francis Lau. vScale: automatic and efficient processor scaling for smp virtual machines. In *EuroSys 2016*, page 2. ACM, 2016.

[72] Xiaoning Ding, Phillip B Gibbons, Michael A Kozuch, and Jianchen Shan. Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, 2014.

[73] Xiang Song, Haibo Chen, Binyu Zang, X SONG, H CHEN, and B ZANG. Characterizing the performance and scalability of many-core applications on virtualized platforms. *Parallel Processing Institute Technical Report Number: FDUPPITR-2010*, 2, 2010.

[74] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 406–418. IEEE Computer Society, 2014.

[75] Artemiy Margaritov, Siddharth Gupta, Rekai Gonzalez-Alberquilla, and Boris Grot. Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–27. IEEE, 2019.

[76] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Addressing fairness in SMT multicores with a progress-aware scheduler. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 187–196. IEEE, 2015.

[77] David Koufaty and Deborah T Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[78] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon. SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms. In *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2012.