Holmes: SMT Interference Diagnosis and CPU Scheduling for Job Co-location

Aidi Pi
epi@uccs.edu
University of Colorado
Colorado Springs

Xiaobo Zhou
xzhou@uccs.edu
University of Colorado
Colorado Springs

Chengzhong Xu
czxu@um.edu.mo
University of Macau

ABSTRACT

Co-location of latency-critical services with best-effort batch jobs is commonly adopted in production systems to increase resource utilization. Although memory and CPU isolation have been extensively studied, we find Simultaneous Multi-Threading (SMT) technology imposes non-trivial *interference on memory access* which jeopardizes efficient co-location and performance assurance of latency-critical services. However, there is not an existing metric to quantitatively measure and lacks a deterministic approach to tackle SMT interference on memory access.

We present Holmes, a user-space approach to SMT interference diagnosis and adaptive CPU scheduling for efficient job co-location in multi-tenant systems. Holmes tackles two challenges: accurately measuring SMT interference on memory access, and efficiently adjusting CPU allocation to achieve low latency and high resource utilization at the same time. It leverages CPU hardware performance events to diagnose SMT interference on memory access and form a metric. It deploys an interference-aware scheduler to adaptively allocate CPU cores to latency-critical services and batch jobs. Experiments with four real-world key-value stores show that compared to a representative CPU isolation approach, Holmes reduces the average (99^{th} percentile) query latency by up to 49.0% (52.3%) for four real-world latency-critical services. It also significantly improves convergence speed, resource utilization, and system throughput.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

Job co-location, latency-critical service, SMT interference

ACM Reference Format:
Aidi Pi, Xiaobo Zhou, and Chengzhong Xu. 2022. Holmes: SMT Interference Diagnosis and CPU Scheduling for Job Co-location. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3502181.3531464

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531464

1 INTRODUCTION

Traditionally, production servers provision resources according to the peak load of latency-critical services, such as key-value store and web search, for meeting their service level objectives (SLOs). Since the peak and average resource consumption of latency-critical services vary significantly over time, this strategy often presents low resource utilization of servers [30, 32, 47, 49]. For instance, SnowFlake system found that the average CPU and memory utilizations on its servers are only ∼ 51% and ∼ 19%, respectively [49]. To improve resource utilization, it is a common practice for production systems to co-locate best-effort batch jobs in the same servers with transiently idle resources, where the shared resources are monitored and isolated such that batch jobs do not interfere with latency-critical services [19, 32, 33, 39, 50, 51, 54].

Motivation. Production systems usually enable SMT to improve server throughput [2–4]. Instructions from different threads can utilize execution units of one physical core in one cycle to improve CPU utilization. However, when instructions from different threads compete for the same execution units, the interference on memory access incurs severe performance degradation on each thread running on a physical core. SMT interference can significantly increase query latency, particularly the tail latency, of co-located latency-critical services and jeopardize their SLOs. Our experiments with real-world latency-critical services and batch jobs show that SMT interference on memory access can cause up to 2× of the average and 2.5× of the 99^{th} percentile query latency compared to when latency-critical services are running alone in a server.

Thus, efficient job co-location demands for a new resource scheduler that takes into account interference of memory access introduced by SMT. Specifically, it should satisfy the following requirements for efficient job co-location in multi-tenant systems.

- Latency-critical services have the high priority under job co-location. Their query latency should be as close to that when the services are running alone in a server. This is the principle of job co-location.
- When the principle is satisfied, the scheduler should seek to improve the server utilization and batch job throughput.
- The scheduler need to be transparent and generally applicable to all applications. To satisfy this, the source code of applications should not be modified.

Limitation of state-of-art approaches. Currently, research on job co-location [19, 32, 33, 50, 54] are either unaware of or neglect the impact of SMT interference on memory access. PerfIso [32] is a representative approach that leverages multicore servers to efficiently share CPU resource between latency-critical services and batch jobs. It enables SMT, but does not take its interference impact into account and it can significantly degrade performance of

latency-critical services. There are studies on improving application performance in servers with SMT enabled [18, 26, 34, 39, 51]. Heracles [39] and Parties [18] are two representative studies that use feedback control approaches to dynamically allocate resources including hyper threads between latency-critical services and batch jobs. A recent study vSMT-IO[34] is a hypervisor-level approach that optimizes I/O-intensive workloads on SMT servers. Nevertheless, none of these studies can quantitatively analyze and tackle SMT interference on memory access for job co-location.

Challenge I: SMT interference measurement. There is no existing quantitative metric to measure SMT interference on memory access. At a first glance, CPU usage might be an indicator to measure the SMT interference since memory access must increase CPU usage. However, a high CPU usage does not necessarily incur a large number of memory accesses and SMT interference since workloads can be computation-intensive. Another naive measurement approach is using a probing process that periodically accesses memory from cores and records the latency. Although the obtained latency could be a quantitative indicator of SMT interference, this approach presents a hard trade-off between measurement accuracy and overhead. Higher probing frequency leads to higher accuracy but also severe competition for shared execution units. The process needs to occupy a chunk of physical memory, which interferes with latency-critical services.

Challenge II: Adaptive core allocation. Co-located batch jobs can use transiently idle resources, but they should not impact performance of latency-critical services. The challenge lies in how to determine the amount of batch job workloads that can be scheduled on a sibling of a processor that serves latency-critical services. Both latency-critical services and batch jobs have three phases during their lifetime: launching, running, and exiting. The scheduler should dynamically adjust the amount of workloads of processors based on phases and profiled memory access latency on each processor. Meanwhile, it needs to guarantee the performance of latency-critical services while keeping the sibling core busy.

Key insights and contributions. In this paper, we present Holmes, a user-space non-intrusive approach for diagnosing SMT interference on memory access and CPU scheduling that takes SMT interference into account for efficient job co-location. Specifically, we target on Intel's implementation of SMT namely Hyper-Threading (HT) which has two hardware threads on a physical core. Unlike memory or CPU usage, HT interference on memory access is not directly exposed by Linux OS and cannot be directly controlled. To tackle this challenge, we leverage hardware performance events (HPEs) provided by Intel processors to diagnose HT interference on memory access. Terms SMT and HT are used interchangeably in this paper. Note that AMD processors support similar hardware performance events called Instruction-Based Sampling.

Identifying the right HPEs for the metric in measuring HT interference is non-trivial since there are hundreds of HPEs reflecting different aspects of performance. We use a statistic approach to select the appropriate HPE and form a metric called *counter value per instruction (VPI)* for HT interference quantification. VPI is a metric to measure load of DRAM access based on HPEs. We will explain the details of VPI for interference diagnosis in Section 3.

To address the second challenge, we design and develop an interference-aware CPU scheduler for efficient job co-location. The scheduler keeps track of CPU status including HT interference and CPU usage on a server with HT enabled. When HT interference is detected, it adaptively adjusts CPU allocation of latency-critical services and batch jobs to mitigate HT interference.

This paper makes the following contributions.

- We conduct in-depth characterization and analysis of SMT interference on memory access, as well as its impact on job co-location. We apply a statistic approach to select appropriate HPEs and use them to calculate a reliable metric (namely VPI) to diagnose HT interference.
- We design an interference-aware CPU scheduler for efficient job co-location in servers with HT enabled, and implement it in user space for transparency to applications and the OS.
- Evaluation shows Holmes achieves latency close to that when services are running alone in a dedicated server, while significantly improving utilization and speedup of convergence.

Experimental methodology and artifact availability. We implement Holmes as a user-space daemon process without modifications to either applications, libraries, or Linux OS. We conduct experiments with four real-world latency-critical services (i.e., Redis, RocksDB, WiredTiger, Memcached) co-located with batch jobs. Results shows that compared to a representative CPU isolation approach PerfIso [32], Holmes reduces the average (99^{th} percentile) query latency for the real-world latency-critical services by up to 49.0% (52.3%) under job co-location. It significantly increases resource utilization and system throughput compared to running latency-critical services alone. Compared to Heracles [39] and Parties [18], Holmes speeds up the convergence on resource allocation by five orders of magnitude. The overhead of Holmes is negligible. Holmes is open source at https://github.com/EddiePi/Holmes.

Limitation of the proposed approach. As many studies in job co-location, Holmes assumes that latency-critical services have bursty traffic such that batch jobs can be allocated with transiently available resources. It is possible that latency-critical services receive consistent high volume of traffic. In this case, batch jobs may be suspended and stop progress for a long time, or even be killed. This situation is acceptable since batch jobs do not have strict SLOs and even be best-effort. In production, there are several solutions if we still want batch jobs to make progress. For example, we could reserve a small dedicated resource pool for batch jobs such that they can make slow progress. Another approach is that batch jobs can be migrated to another machines with more resources in the cluster.

In the rest of the paper, Section 2 presents SMT background and motivational studies. Sections 3 and 4 describe SMT interference measurement and Holmes design. Section 5 presents the implementation. Section 6 evaluates Holmes. We discuss related work in Section 7 and conclude in Section 8.

2 INTERFERENCE VERIFICATION

2.1 Simultaneous Multi-Threading

We show the execution process of multi-threading and Simultaneous Multi-Threading (SMT) in Figure 1, considering there are two

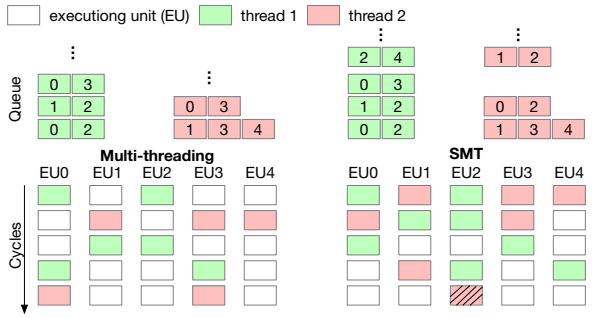


Figure 1: Example of multi-threading and simultaneous multi-threading with 2 threads and 5 execution units.

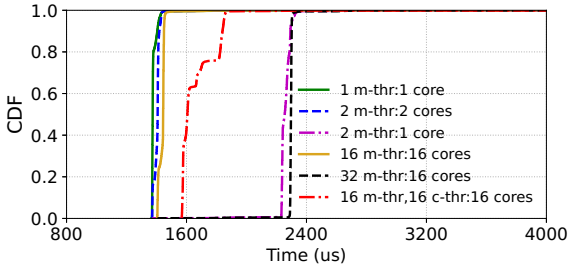


Figure 2: Memory access latency from different sources.

threads issuing instructions. Instructions are placed in a queue, and the number in each instruction indicate its required execution units (EU). For example, the first instruction from thread 1 has numbers "0, 2", indicating it uses execution unit EU0 and EU2. In a multi-threading processor, only one thread can run on execution units at a time. When the thread encounter a long waiting event or uses up its quota, the processor switches to another thread. In contrast, SMT allows multiple hardware threads run on the execution units of the same physical core in the same cycle. We target Intel's implementation of SMT, known as HT technology. HT implements a 2-way SMT which makes a physical core appears as two processors to an operating system. Two thread contexts can be simultaneously kept on a HT core while sharing the same set of execution units. HT expects that combining and executing instructions from two threads increases the utilization of a physical core. However, when there are two instructions compete for the same execution unit, one will be delayed. In Figure 1, for example, the last instruction from thread 2 is delayed on EU2 (shaded rectangle) since thread 1 is using EU2.

2.2 Why SMT Interference?

Previous studies [43–45] showed that memory controller and bandwidth congestion are the main bottleneck for memory access latency. We find that these bottleneck have been well addressed on a modern CPU. It is HT interference that degrades memory access latency.

Micro benchmark. We use a case study with a micro benchmark to illustrate the sources of memory access latency. We use the same server configuration as that in Section 6. The benchmark has

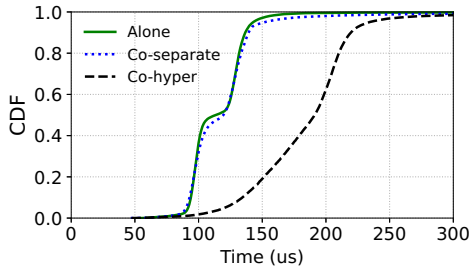


Figure 3: Latency of Redis under the different settings.

a configurable number of threads pinned onto individual logical CPUs. We run two kinds of threads: 1) threads continuously send memory requests to access a random 1MB memory block out of 600MB data (denoted as m-thread), 2) threads run a computation-intensive workload executing floating point operations (denoted as c-thread). To identify the sources of memory access latency, we evaluate the latency of m-threads accessing 1MB memory block in six cases. For the first five cases, all threads are m-threads. We use the following five thread allocation: 1) 1 thread on 1 core, 2) 2 threads on 2 cores, 3) 2 threads on 2 logical CPUs of the same core, 4) 16 threads on 16 cores, and 5) 32 threads on 32 logical CPUs of 16 cores. In case 6), 16 m-threads are allocated on 16 logical CPUs and another 16 c-threads are allocated on the siblings of the 16 logical CPUs. Case 1) is used as the baseline. 2), 3) and 6) are used to inspect the impact of Hyper-Threading. 4) and 5) are used to inspect whether memory controller or bandwidth is the bottleneck.

Figure 2 shows the CDF of the memory access latency in the six scenarios. Cases where memory is accessed from individual physical cores render an average latency about $1,400\mu s$ regardless of the number of accessing threads. In these cases, memory controller congestion or memory bandwidth congestion has little impact on memory access latency since cases 1), 2) and 4) have almost the same performance. Case 3) case 5) both use HT while case 5) has much more requests sending from 32 threads. If case 5) is bottlenecked by memory bandwidth, it should render an even higher latency than that of case 3), which is however not the case. In these two cases, threads are scheduled on siblings using HT render a higher average latency, about $2,300\mu s$, compared to those without Hyper-Threading. By comparing results of case 5) and case 6), we reveal that computational-intensive workload can degrade the performance of memory access on its sibling thread, while its impact is much less than that of memory accessing workloads.

CPU scheduling gets even more complicated when latency-critical services are co-located with batch jobs. Since latency-critical services have bursts of incoming queries [32], statically allocating fixed amount of resource usually results in either sub-optimal performance or resource wastage.

Job co-location with a real-world service. We use Redis, a real-world latency-critical service, to illustrate the significant impact of HT interference on query latency. Redis is running under three settings. 1) Alone: Redis runs alone with HT enabled. 2) Co-separate: Redis runs with batch jobs and they use separate physical cores. 3) Co-hyper: Redis runs with batch jobs and the batch jobs are allowed to use siblings of Redis cores. Redis runs with the same

Table 1: The Candidate HPEs.

Name	Description	Event #	Corr
CYCLES_ L3_MISS	Cycles while L3 cache miss demand load is outstanding.	0x02A3	-0.1748
STALLS_ L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	0x06A3	0.9992
CYCLES_ MEM_ANY	Cycles when memory subsystem has an outstanding load.	0x10A3	0.9997
STALLS_ MEM_ANY	Execution stalls when memory subsystem has outstanding load.	0x14A3	0.9999

number of threads in the three cases. We use the Spark Kmeans workload from HiBench [31] as the batch job. In the two co-location settings, threads of Redis and threads of the batch job are pinned on different logical CPUs. We use workload-a from cloud serving benchmark YCSB [23] to generate queries to Redis.

Figure 3 shows the CDF of the latency of Redis queries under different settings. Settings Alone and Co-separate render similar latency for Redis queries. However, the latency is significantly prolonged with Co-hyper setting where queries are affected by HT interference. In Co-hyper, the average (99^{th} percentile) query latency of Redis with HT is 2.0× (1.3×) as high as that of Co-separate. **[Summary]** We have identified that HT interference degrades memory access latency for both a micro benchmark and a real-world service, while memory controller/bandwidth congestion has little impact on their latency. Since latency-critical services and batch jobs both frequently access memory, the co-location could significantly degrade performance of latency-critical services. Though, HT interference can be qualitatively identified, there is no efficient approach that can quantitatively and accurately measure it.

3 INTERFERENCE DIAGNOSIS

3.1 Finding the Metric

We identify a set of hardware performance events (HPEs) provided by Intel processors and apply a statistic approach to select the appropriate HPE to form a metric. We then leverage upon the metric to accurately diagnose HT interference on memory access.

We choose four candidate HPEs as they are related to LLC miss and the memory subsystem. Table 1 gives their names, description, event numbers, and correlation. To evaluate the correlation between the counter value of each candidate HPE and the memory access latency, we deploy a measurement program that runs on either one or both threads to continuously send fixed-sized (e.g. 1MB) memory requests from the same cores to DRAM for a unit time (e.g. one second). When one thread is running, we change the request sending rate (i.e. requests per second, RPS) ranging from 5,000 RPS to its maximum rate around 74,000 RPS with 5,000 as the step size. When two threads are running, we pin the two threads on the siblings of one physical core. We fix one thread to its maximum possible RPS (Figure 4(b)) and change the RPS of the other thread from 5,000 to its maximum rate around 45,000 RPS with 5,000 as the step size Figure 4(c)). We make sure that the requested data do not reside in any layer of CPU caches.

Counter value per second. We record the counter value of each candidate HPE from the logical CPU that hosts the measurement program at run time. Initially, we intend to use counter value per second as the metric to measure HT interference on memory access. However, this method is not effective if a logical CPU is not fully loaded. For example, when a thread on a processor sends requests with 5,000 RPS and its sibling processor is fully loaded, the thread experiences a long memory access latency while the recorded per-second counter value is relatively small since only 5,000 requests are sent. Such a small counter value does not really reflect the high memory access latency. The scenario happens even with a more fine-grained time unit.

Counter value per instruction (VPI). In order to accurately model DRAM access latency, the actual load of DRAM access on a processor needs to be obtained. We achieve this by recording the sum of the number of instructions `LOAD` and `STORE` and divide the counter values by the sum, as shown in Equation 1. In this way, we calculate the average counter values per DRAM access instruction VPI_{event}. We normalize the average memory access latency and VPI_{event} of each HPE to their own maximum values.

$$VPI_{event} = \frac{CounterValue_{event}}{Num_{LOAD} + Num_{STORE}} \quad (1)$$

Figure 4(a) shows the memory access latency and VPIs of the four HPE events when we increase RPS from 5,000 to 74,000 in the one-thread configuration. It clearly shows that the memory access latency remains almost unchanged with one thread regardless of RPS. Figures 4(b) and 4(c) show the memory access latency and VPIs under the two-thread configuration. In specific, Figure 4(b) shows the thread always using the maximum RPS and Figure 4(c) shows the thread using various RPS. For the thread with various RPS, its memory request latency remain unchanged regardless of its RPS. For the thread with the maximum RPS, its maximum RPS decreases from $\sim$ 70,000 to $\sim$ 45,000 with the increasing RPS on its sibling thread. Its memory access latency also increases. Among the four HPEs, the VPI of CYCLES_MEM_ANY (0x10A3) and STALLS_MEM_ANY (0x14A3) presents almost the identical trend as the memory access latency does.

We use Pearson's correlation coefficient to statistically quantify the correlation between the memory access latency and the value of each HPE, as shown in `Corr` column of Table 1. A correlation score that is close to 1 or -1 means the strongest correlation between the two metrics. In Table 1, event STALLS_MEM_ANY (0x14A3) has the highest correlation score (0.9999) among the four HPEs, suggesting a strong positive correlation with the memory access latency. HPE CYCLE_L3_MISS (0x02A3) has relatively low correlation score. We notice that HPEs STALLS_L3_MISS (0x06A3) and CYCLES_MEM_ANY (0x10A3) also present high positive correlation scores, but they are slightly lower than that of HPE STALLS_MEM_ANY (0x14A3). Therefore, we use the counter value of HPE STALLS_MEM_ANY (0x14A3) according to Equation 1, that is VPI_{0x14A3}, as the metric to diagnose HT interference between siblings of a physical cores.

3.2 Effectiveness of the Metric

We conduct experimentation to examine that VPI_{0x14A3} is an effective metric in measuring HT interference on latency-critical services. We test four real-world latency-critical services Redis [9], RocksDB [10], WiredTiger [11], and Memcached [8]. Workload-a from YCSB [23] is used to generate requests to the latency-critical

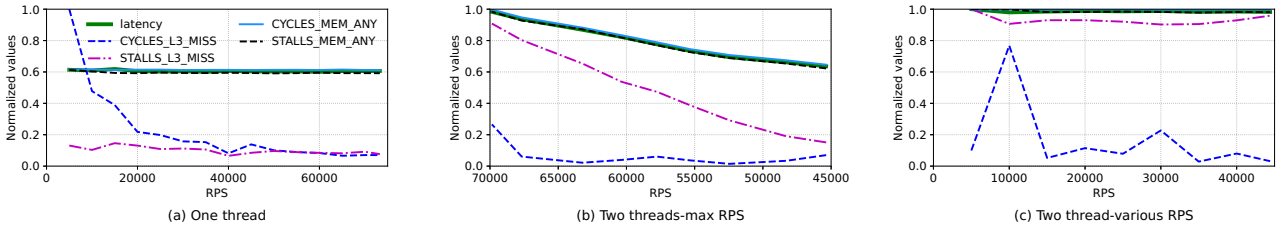


Figure 4: The normalized memory access latency and value of HPEs under different thread configurations.

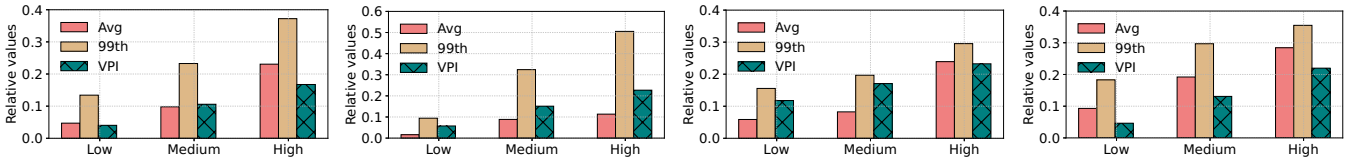


Figure 5: The correlation between VPI and the normalized average and 99^{th} percentile latency of latency-critical services.

services. We develop a program that can access memory with configurable request rate (i.e. request per second, RPS). For each latency-critical service, we pin their threads on four logical CPUs and pin the threads of the memory access program on the siblings of the four logical CPUs. In the experiment, we use three RPS settings 20,000 (Low), 40,000 (Medium), and 60,000 (High). For each setting, we start the memory access program and YCSB concurrently. We also run YCSB workload alone without the memory access program as the baseline (Alone) for normalization. We sum VPI_{0x14A3} on the four logical CPUs during the execution.

We normalize both VPI and latency of the services to those in Alone setting by using $\frac{V-V_{Alone}}{V_{Alone}}$ (V stands for either VPI or latency of the services.). For example, an avg bar with value 0.3 indicates that the average latency of the service is 30% higher than that under Alone setting. Figure 5 shows the normalized average latency, 99^{th} percentile latency, and VPI_{0x14A3} for the four services under each setting. As the load increases, the growth patterns of latency and VPI are very similar, which indicates that VPI_{0x14A3} is an effective metric that can quantitatively reflect the latency of the services.

4 HOLMES DESIGN

4.1 Overview

Figure 6 presents the closed-loop design of Holmes. The metric monitor keeps track of the status of both latency-critical services and best-effort batch jobs, as well as the resource usage of the server. Holmes diagnoses HT interference on memory access based on the quantification approach with the selected HPE. It then adaptively adjusts CPU core allocation for latency-critical services and batch jobs in a shared server based on process and system status. The CPU scheduler communicates with Linux kernel and adjusts core allocation at runtime using an interference-aware scheduler, which aims to achieve low latency for latency-critical services and improve server resource utilization and throughput. In return, the adjusted core allocation affects the performance metrics in the system.

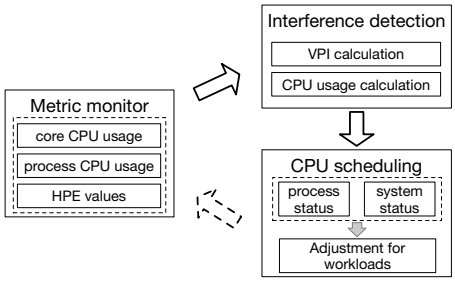


Figure 6: An overview of Holmes architecture.

4.2 Metric Monitor

The metric monitor thread is periodically invoked to collect the information of CPU usage and CPU/thread status of running workloads.

CPU resource usage. The monitor thread collects both CPU usage and VPI_{0x14A3} of each virtual processor. It stores the metrics in an array for all cores in a server. Holmes maintains logical-processor-to-core mappings. The collected processor metrics are aggregated per core by accumulating both processor metrics on that core. As a result, Holmes presents the overall CPU usage and the overall VPI_{0x14A3} of each core in the server.

Process status. For processes of latency-critical services and those of batch jobs, the monitor thread collects their CPU/ thread usages and thread-to-processor mappings. Since latency-critical services usually are long running, their PIDs are provided to Holmes by the system administrator upon service initialization. In contrast, batch jobs are usually launched by a resource scheduler, such as Apache Yarn [48], and finish at arbitrary time. To detect batch jobs when they start, we configure Yarn to launch batch job processes in Linux containers. Each container uses its own files in the *cgroup* file system to achieve resource allocation and isolation. Holmes monitors directories in the *cgroup* file system to detect batch jobs.

Table 2: Terminologies in Holmes.

Terminologies	Description
Reserved CPU	A logical CPU reserved for latency-critical services. Batch jobs are not allowed to run on it.
Non-reserved CPU	A logical CPU other than the reserved CPUs. All processes can run on them.
LC CPU	A logical CPU hosting latency-critical services.
LC-sibling CPU	The sibling CPU of a LC CPU.
Non-sibling CPU	The CPUs that their siblings are not hosting latency-critical services.

Holmes determines whether a latency-critical service is serving query traffic by monitoring its CPU usage. Most queries of a latency-critical service are related to memory access. Some services have background management threads that are responsible for data merge or compaction operations which are also memory intensive. When these operations are on the fly, they incur CPU usage.

4.3 CPU Scheduler

The CPU scheduler periodically reads the information collected by the metric monitor and adaptively adjusts core allocations using an interference-aware scheduling algorithm for co-located latency-critical services and batch jobs. The algorithm prioritizes latency-critical services by reserving a small number of logical CPUs for bursts of query traffic. When latency-critical services do not have traffic, it allows batch jobs to utilize the other non-reserved logical CPUs. Though this may incur HT interference on batch jobs, it is not considered as an issue since batch jobs do not have a hard deadline for completion. Further, by leveraging HT, batch jobs can achieve higher throughput. When latency-critical services are serving traffic, the scheduler adjusts core allocations for both types of workloads. We describe the CPU scheduling algorithm corresponding to the three phases of the lifetime of a process, launching, running and exiting. Table 2 describes the terminologies used in Holmes.

Algorithm 1 Process launching.

```
1: pid_set_lss: A set of process IDs of latency sensitive services;
2: pid_set_batch: A set of process IDs of batch jobs;
3: pid = launched process id;
4: if pid_set_lss contains pid then
5:     ALLOCATE(rsv_CPUs, pid);
6: else
7:     ALLOCATE(non_rsv_CPUs, pid);
8: end if
```

Process launching. The process launching procedure is shown in Algorithm 1. Upon launch of a latency-critical service, Holmes allocates reserved CPUs to the service. Upon launch of containers for a batch job, Holmes allocates non-reserved CPUs to them. Among these non-reserved CPUs, Holmes first chooses the ones from non-sibling CPUs. The number of CPUs is specified by configuration files of the batch job. When non-sibling CPUs are busy, Holmes chooses CPUs from all the non-reserved CPUs for the containers as long as the overall VPI_0x14A3 of their sibling CPUs is less than a threshold E. By this design, performance of latency-critical services is not affected by batch jobs at launching time. Meanwhile, LC-sibling CPUs could be utilized by batch jobs when necessary.

Process running. The procedure during process running is shown in Algorithm 2. Holmes checks whether a latency-critical service is serving query traffic by monitoring its CPU usage. When it is serving traffic, the usage of the reserved CPUs and VPI_{0x14A3} of the CPUs hosting the service must increase. If there are co-located batch jobs on siblings of these CPUs, Holmes adjusts core allocation based on these two metrics accordingly.

Algorithm 2 Process running.

```
1: for each LC_CPU do
2:     for pid in pid_set_batch do
3:         while VPI_0x14A3(LC_CPU) ≥ E do
4:             if sibling_CPU of LC_CPU has pid then
5:                 DEALLOCATE(sibling_CPU, pid);
6:                 if non_sibling_CPUs.available then
7:                     ALLOCATE(non_sibling_CPUs, pid);
8:                 end if
9:             end if
10:        end while
11:    end for
12:    if VPI_0x14A3(LC_CPU) ≤ E for S seconds then
13:        pid ← CHOOSE_ONE(pid_set_batch)
14:        ALLOCATE(sibling_CPU, pid)
15:    end if
16: end for
17: while reserved_CPUs.usage > T% do
18:    new_CPU ← GET_OR_DEPRIVE(all_CPUs);
19:    reserved_CPUs.add(new_CPU);
20: end while
```

When VPI_{0x14A3} is greater than or equal to threshold E, HT interference is detected. In this case, Holmes deallocates the sibling CPUs from the containers of the batch jobs. When VPI_{0x14A3} of the LC CPU drops below threshold E for S seconds, Holmes re-allocates the sibling CPUs to the containers of the batch jobs. Meanwhile, Holmes seeks to allocate non-sibling CPUs to the containers of the batch jobs. This could happen when the containers of the batch jobs on a non-sibling CPU finish. By deallocating the CPUs from the containers of the batch jobs, VPI_{0x14A3} is reduced accordingly. Although processing of the batch jobs is slowed down, their execution progress are preserved.

When the average usage of the reserved CPUs reaches T ($0<T<1$) of their capacity but VPI_{0x14A3} is less than E, the capacity of the reserved CPUs is not enough to serve the latency-critical service. For example, we initially assign four reserved CPUs to the latency-critical service and set T to 80%. When the CPU usage of the four cores increases beyond 320% ($4 * 80\%$), Holmes starts an expansion procedure. It could happen when the latency-critical service creates more active threads than the number of the initial reserved CPUs. Holmes adds one CPU at a time until the capacity is enough to serve the latency-critical service. The chosen CPU is not the sibling of the current LC CPUs. At the same time, if any batch job is running on the sibling of the chosen CPU, Holmes deallocates the CPU from batch jobs to ensure the performance of latency-critical service.

Algorithm 3 Process exiting.

```
1: if lss exits then
2:     for pid in pid_set_batch do
3:         ALLOCATE(sibling_CPUs, pid);
4:     end for
5: end if
6: if batch on non_sibling_CPUs exits then
7:     for pid in pid_set_batch do
8:         REALLOCATE(non_sibling_CPUs, pid);
9:     end for
10: end if
```

Process exiting. The procedure is shown in Algorithm 3. There are two situations of process exiting that need to be managed: 1)

exiting of containers of the batch jobs on non-sibling CPUs, and 2) traffic of the latency-critical service is over. Upon exiting of containers of a batch job on non-sibling CPUs, Holmes examines whether any sibling CPU is hosting containers of the batch jobs. If so, Holmes migrates some of them onto non-sibling CPUs. When the latency-critical service finishes traffic serving, Holmes allocates sibling CPUs to containers of the batch jobs whose CPUs are previously deprived. By this design, it improves the CPU resource utilization of a server while guaranteeing resources for latency-critical services.

5 IMPLEMENTATION

We implement Holmes in $\sim 3,000$ lines of C++ code. Holmes runs as a daemon process with root privilege. We empirically set the invocation interval at 50 μs for both the metric monitor thread and CPU scheduling thread. The interval is set based on the fact that latency-critical services usually have a per-query respond time at several hundreds of microseconds. The 50 μs invocation interval is short enough for fast HT interference detection and CPU scheduling while maintaining low overhead. We set the number of reserved cores to four in our 32-core servers. We set the deallocation threshold E for batch jobs at 40. This is a rather strict value since we expect that the processors of batch jobs can be deallocated at an early stage, resolving HT interference. We empirically set the CPU usage threshold T of latency-critical services for core expansion at 80%. In this case, it allows 20% CPU quota for a burst of queries before Holmes allocates more cores to latency-critical services. When a latency-critical service is launched, the system administrator specifies its PID to Holmes.

Linux APIs. Holmes communicates with Linux OS for two tasks: 1) collecting HPE counter values, and 2) allocating cores to processes. Holmes uses system call *perf_event_open* to collect HPE values provided by Intel processors at runtime. Holmes allocates cores for threads by invoking system call *sched_setaffinity*. Note that AMD processors provide similar technology called Instruction-Based Sampling [1].

Batch jobs management. We use Apache Yarn [48] to manage batch jobs. A batch job can be divided and launched in multiple Linux containers [7]. We modify the source code of Yarn NodeManager to launch batch jobs with a set of specified cores, which makes sure that batch jobs are not allocated with the cores reserved for latency-critical services. The modification takes less than 10 lines of code in Yarn. We create a parent cgroup directory to manage all Linux containers for batch jobs. Each batch job container is associated with an individual directory. Holmes keeps track of the liveliness and resource consumption for batch jobs by scanning all the cgroup directories.

6 EVALUATION

6.1 Evaluation Setup

We use two servers in the experimentation. Each server has two Intel Xeon Gold 6143 CPUs (32 cores per CPU), 256 GB DRAM, and 512 GB SSD. We use four real-world latency-critical services to evaluate the performance of Holmes: Redis-5.0.5 [9], Memcached-1.5.22 [8], RocksDB-6.0.0 [10], and WiredTiger-3.2.1 [11]. Redis and Memcached are two in-memory key-value (KV) stores. RocksDB is a state-of-the-art disk-based persistent KV store based on Long-Structured Merge Tree (LSM Tree). WiredTiger is the latest disk-based persistent KV storage engine for MongoDB. Both disk-based KV stores keep an in-memory cache. When they access data stored in files, Linux OS also keeps page cache for the recently accessed files. Therefore, Holmes is applicable to disk-based KV stores.

We use Yarn in Hadoop-2.9.2 [5] as the job scheduler and Spark-2.4.5 [53] as the data analytics engine for batch jobs. One server hosts latency-critical services and batch jobs. The other serves as the client of the latency-critical services and Yarn's master node.

Performance of Holmes is evaluated in five metrics: 1) query latency and SLO violation of latency-critical services, 2) server throughput in terms of CPU utilization and the number of completed batch jobs, 3) parameter sensitivity of Holmes, 4) convergence speed on resource allocation, and 4) overhead of Holmes.

We use the following configurations to generate job co-location on the server, and use it for latency (§ 6.2) and throughput (§ 6.3) evaluation. We submit workloads in YCSB [23] to generate bursty query traffic for latency-critical services. Each bundle of bursty traffic lasts for 60s∼90s with an interval ranging from 5s∼10s. Both traffic time periods and interval periods agree to Poisson distribution. Similar to other studies [19, 42], we continuously submit multiple concurrent workloads in HiBench-6.0 [31] as batch jobs. Each batch job lasts for around three minutes. After batch jobs are submitted, all processors on the server are allocated to batch jobs except for the reserved processors for latency-critical services. We make sure there is no memory pressure on the server by constraining the memory limit of containers of batch jobs.

We conduct experiments in three settings.

- **Alone.** Latency-critical services run in a dedicated server without job co-location. It is an ideal scenario for the services, but the server suffers from low resource utilization.
- **PerfIso.** Latency-critical services are co-located with batch jobs. Naive CPU isolation by representative PerfIso [32] is enabled to dynamically adjust processor allocation.
- **Holmes.** Latency-critical services are co-located with batch jobs. Holmes is enabled to dynamically adjust core allocation to mitigate the impact of SMT interference and CPU interference on the latency-critical services.

6.2 Query Latency Reduction

We examine query latency when latency-critical services serve three representative workloads in YCSB under the settings of Alone, Holmes, and PerfIso. Workload-a is a write-heavy workload consisting of 50% read and 50% update queries. Workload-b is a read-heavy workload consisting of 95% read and 5% update queries. Workload-e is a scan-heavy workload consisting of 95% scan and 5% insert queries. Note that there is no workload-e for Memcached service since it does not support scan operations. We show the CDF of query latency of the workloads for the four latency-critical services running under the three settings in Figures 7 to 10.

Redis and Memcached. They are two popular in-memory KV stores. As shown from Figure 7 to Figure 10, naive isolation due to PerfIso significantly prolongs the query latency for all workloads at each percentile. For Redis service, Holmes reduces the average (99^{th} percentile) query latency by 49.0% (35.2%), 40.7% (11.7%), and 28.1%

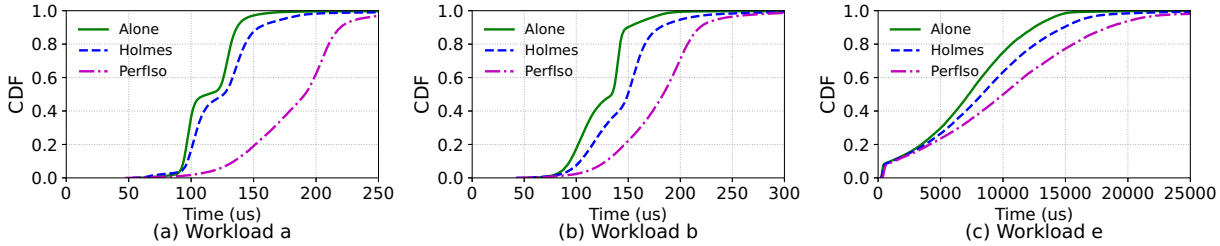


Figure 7: The CDF of query latency of Redis service under three workload settings.

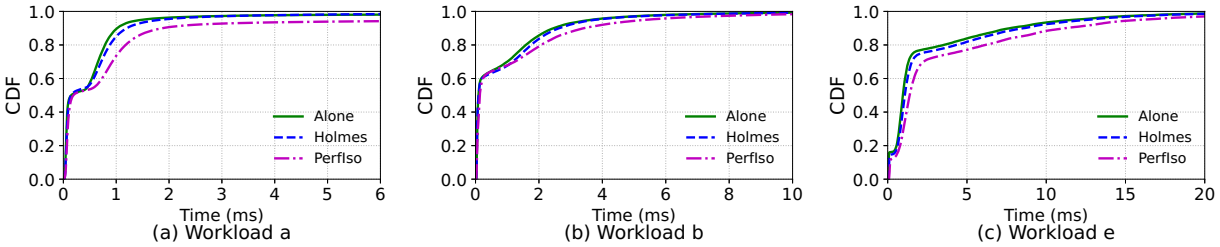


Figure 8: The CDF of query latency of RocksDB service under three workload settings.

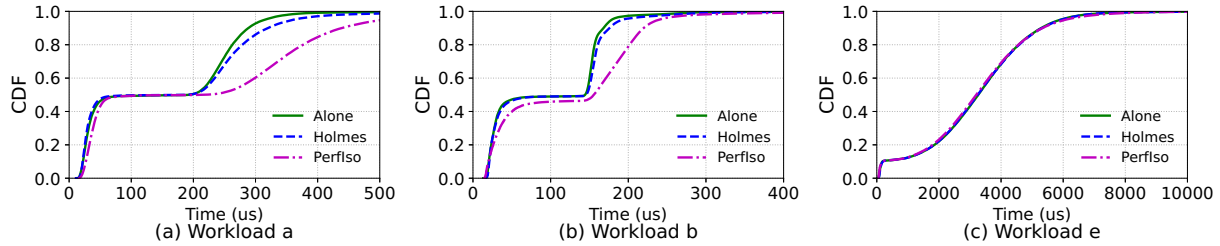


Figure 9: The CDF of query latency of WiredTiger service under three workload settings.

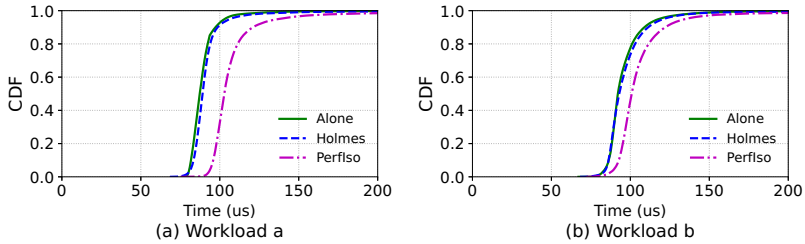
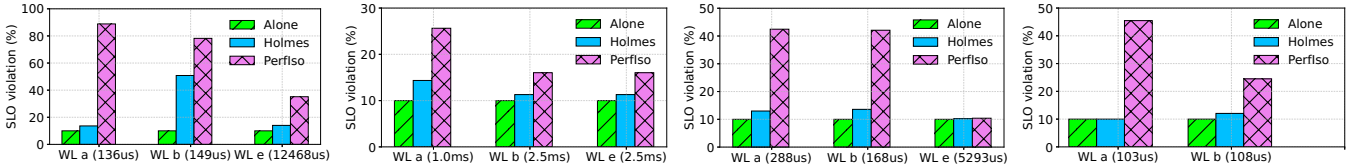


Figure 10: The CDF of query latency of Memcached service under two workload settings.

(49.3%) for the three workloads, respectively. We observe a very long tail (99^{th} percentile) of query latency in workload-b of Redis due to PerfIso. It suggests that HT interference has significant impact on the read-heavy workload for Redis. For Memcached service, compared to PerfIso, Holmes reduces the average (99^{th} percentile) latency by 16.9% (52.3%) and 9.5% (39.2%) for the two workloads, respectively. It achieves almost identical query latency as Alone setting does for both workload-a and workload-b.

RocksDB and WiredTiger. They are two disk-based KV stores. As shown in Figure 8 and Figure 9, Holmes achieves almost identical query latency as Alone setting does for both services. For RocksDB service, queries have a long tail of latency even if the service is running under Alone setting. The long tail further deteriorates when HT interference presents. Compared to PerfIso, Holmes reduces the average (99^{th} percentile) query latency by 44.2% (28.9%), 18.9% (28.7%), and 25.0% (18.8%) for the three workloads, respectively. For WiredTiger service, compared to PerfIso, Holmes reduces the average (99^{th} percentile) query latency by 21.6% (30.1%) and 19.4% (21.7%) for workload-a and workload-b, respectively. As for workload-e, the query latency under the three settings is almost identical, which implies that the workload is insensitive to HT interference.



(a) Redis SLO violation (b) RockDB SLO violation (c) WiredTiger SLO violation (d) Memcached SLO violation

Figure 11: The SLO violation of four latency-critical services under three workload settings.

We notice that for both disk-based KV stores, the CDF curve is a stair-like shape. For example, about 50% of the queries in workload-a of RocksDB have latency around 200μs while the other 50% of the queries have latency raging from 500μs to several milliseconds.

There are two reasons that cause the stair-like CDF curve. 1) For workload-a, half of the queries are of update and the other half are of read. Since both services use an asynchronous technique for update queries, one half of the queries (update) return quickly. For the other half of the queries (read), the services need to access memory or disks before returning data. Thus, the read queries have much slower response than the update queries. 2) Workload-b and workload-e are both read intensive. The services have low latency when the queries hit in-memory cache for the queried data, and have high latency when the queries need to access disks. For disk-based KV stores, the results show that all three settings cause similar query latency in low percentiles. HT interference has more significant impact on query latency in high percentiles.

SLO violation. Figure 11 shows the ratios of SLO violation due to Alone, Holmes and PerfIso for the four latency-critical services, respectively. Latency-critical services like web search commonly distribute requests across many servers, and the end-to-end response time is determined by the slowest individual latency [13, 24, 27, 55]. There is not a magic value that defines the SLO of each service. We adopt the 90^{th} percentile latency under Alone as the SLO. In Redis, the SLOs are set to 136μs, 149μs and 12,468μs for workload-a, -b and -e, respectively. These are rather strict values as only 10% SLO violations are allowed under Alone. Compared to Alone, Holmes achieves a similar SLO violation ratio in most cases, especially for disk-based services (i.e., RocksDB and WiredTiger). One exception is workload-b with Redis. Holmes incurs a violation ratio of 50.8% because workload-b is very sensitive to HT interference and parameter setting. PerfIso causes significantly worse SLO violation ratios in all cases. Its SLO violation ratios are usually above 25%, and around 90% in the worst case.

[Summary] Experiments show that Holmes reduces the average (99^{th} percentile) latency by up to 49.3% (52.3%) for the four in-memory KV stores. Since disk-based KV stores have memory cache, Holmes can reduce their average (99^{th} percentile) latency by up to 44.2% (30.1%). In most cases, Holmes achieves very similar query latency and SLO violation ratio for the latency-critical services as Alone does. We notice that the latency of Redis due to Holmes still has some degradation compared to the Alone case. The possible reason is that Redis uses a single thread to serve all user requests. When requests are delayed on the thread, there is no other thread to dispatch the requests, resulting in longer latency.

6.3 Server Throughput Improvement

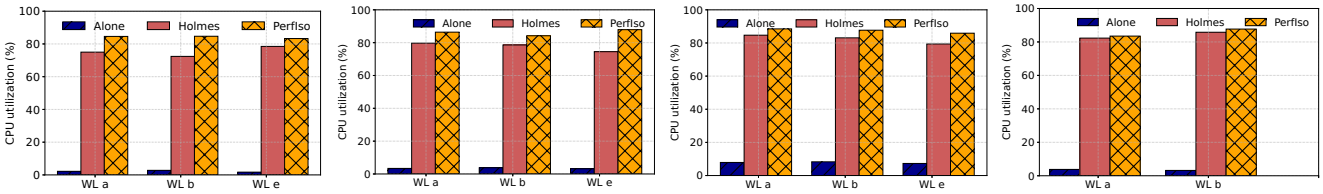
We examine the server throughput when running the four latency-critical services with three workloads in one hour, under Alone, Holmes, and PerfIso. The metrics are 1) CPU utilization, and 2) the number of completed batch jobs.

CPU utilization. Figure 12 shows the average CPU utilization with the four latency-critical services under the three settings Alone, Holmes, and PerfIso. There is no significant difference in the utilization due to different workloads. Overall, Holmes achieves the average CPU utilization 72.4% ~ 85.8% while PerfIso achieves the average CPU utilization 83.4% ~ 88.5%, both under job co-location. Compared to Alone, both Holmes and PerfIso significantly improve CPU utilization of the server. PerfIso slightly outperforms Holmes in CPU utilization, it however significantly violates the SLO of latency-critical services, the principle of job co-location. Note that it is the interference-aware CPU scheduling of Holmes that assures query latency of latency-critical services under job co-location close to that when the services are running alone in a server.

VPI value. Figure 13 offers a microscopic view of the average VPI value on the LC CPUs at runtime when Rocksdb is serving workload-a under PerfIso, Holmes and Alone. Other latency-critical services and workloads have similar results. Alone has the most stable VPI value. PerfIso renders the highest VPI value and fluctuation. Holmes leads to lower and more stable VPI value compared to that due to PerfIso. The reason is that Holmes deallocates CPUs of batch jobs when Rocksdb is serving traffic, and restores CPUs when the traffic is over. Thus, the VPI value on the LC CPUs is less affected by batch jobs compared to PerfIso.

Memory utilization. The server memory utilization under the three settings does not change much. For Alone, the memory utilization stabilizes around 2GB for Redis and Memcached, and around 1GB for RocksDB and WiredTiger. For PerfIso and Holmes, the memory utilization stabilizes around 144 GB for all four latency-critical services co-located with batch jobs. There are two reasons for the stable memory utilization. 1) Latency-critical services use memory as data storage or data cache. Their memory consumption does not change much unless there are more data inserted into the services. 2) Each container of a batch job is configured with a fixed size of memory. Its memory consumption does not change unless the memory size of the containers is changed.

Number of completed jobs. Table 3 gives the number of completed batch jobs in one hour when Redis serves workload-a in the three settings. With PerfIso, 78 batch jobs are completed. With Holmes, 73 batch jobs are completed. Holmes adaptively adjust LC-sibling CPU allocation for batch jobs which slows down batch



(a) Redis CPU utilization (b) RockDB CPU utilization (c) WiredTiger CPU utilization (d) Memcached CPU utilization

Figure 12: CPU utilization with four latency-critical services under three settings (Alone, Holmes, PerfIso).

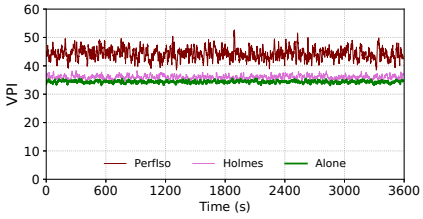


Figure 13: VPI under three settings for Rocksdb.

Table 3: Throughput Comparison of Three Settings.

settings	avg. CPU usage	# finished batch jobs
Co-location with PerfIso	84.6%	78
Co-location with Holmes	75.0%	73
Alone	1.1%	0

job processing in exchange for latency assurance of latency-critical services, the principle of job co-location.

[Summary] Holmes significantly improves CPU utilization and system throughput in a multi-tenant system, compared to a dedicated system where latency-critical services are running alone. It significantly outperforms PerfIso in query latency assurance for latency-critical services.

6.4 Parameter Sensitivity

We examine the impact of threshold E of VPI_{0x14A3} on query latency of latency-critical services. In Holmes, threshold E determines when sibling cores of containers of each batch job is disabled. It is a trade-off between CPU utilization and meeting SLO of a latency-critical service. A lower value yields more disabled processors and thus lower query latency for meeting the SLOs, but also fewer processors for batch jobs that reduces the job throughput and CPU utilization of the server. For efficient job co-location, the primary goal of parameter tuning is to achieve low latency for latency-critical services while the secondary goal is to improve server utilization.

We use workload-a in YCSB and change E from 40 to 80 with step size 10. We normalize the query latency due to Holmes to that due to Alone for the four real-workload services. Figure 14 shows the normalized latency on the average latency and at four specific percentiles. It shows that E with value of 40 renders almost similar results as Alone does in most cases. This value yields the latency similar to those in Alone for Redis, WiredTiger and Memcached at each percentile. For RocksDB, it yields slightly worse results than those Alone does. For RocksDB, Holmes can tune for a lower value for E (e.g. 10) to deallocate cores of batch jobs more promptly.

[Parameter Tuning] When users tune parameter E, there are multiple factors to take into account, such as SLO of a latency-critical service, the type of batch jobs, hardware configuration, etc. It may result in a higher value of E if a latency-critical service has a loose SLO and server utilization is more important, or a lower value if SLO of the service is not allowed to be compromised.

Table 4: Convergence speed of four approaches.

Approach	Convergence speed
Heracles	30s
Parties	10-20s
Caladan	$20\mu s$
Holmes	$50-100\mu s$

6.5 Convergence Speed on Resource Allocation

Three approaches [18, 27, 39] consider HT in tackling interference caused by job co-location. Table 4 summarizes the typical convergence speed of the three approaches and Holmes. Heracles [39] and Parties [18] reach convergence after tens of seconds. Given that the typical query latency of latency-critical services is usually hundreds of microseconds to tens of milliseconds, this second-level convergence speed can be too slow to reduce SLO violation for latency-critical services. Holmes speeds up the convergence on the resource allocation by five orders of magnitude. We note that Caladan [27] achieves even faster convergence at around $20\mu s$. Caladan scheduler requires modification to the Linux Kernel source, while Holmes is a user-space approach without modification to libraries or Linux kernel. These are two complementary approaches for mitigating HT interference for latency-critical services.

6.6 Overhead

We analyze the overhead of Holmes. Holmes introduces about 1.3% $\sim$ 3% CPU usage depending on whether the scheduling threads are active in management operations. It occupies about 2MB memory at runtime, which is negligible compared to the memory capacity of a DRAM node. We suggest launching Holmes on a separate core to minimize its interference with latency-critical services.

6.7 Discussions

HPE dependency. The HPEs selected in Holmes can be observed from all processors using Intel 64 and IA-32 architectures. Job co-location with Holmes on servers with AMD processors can adopt

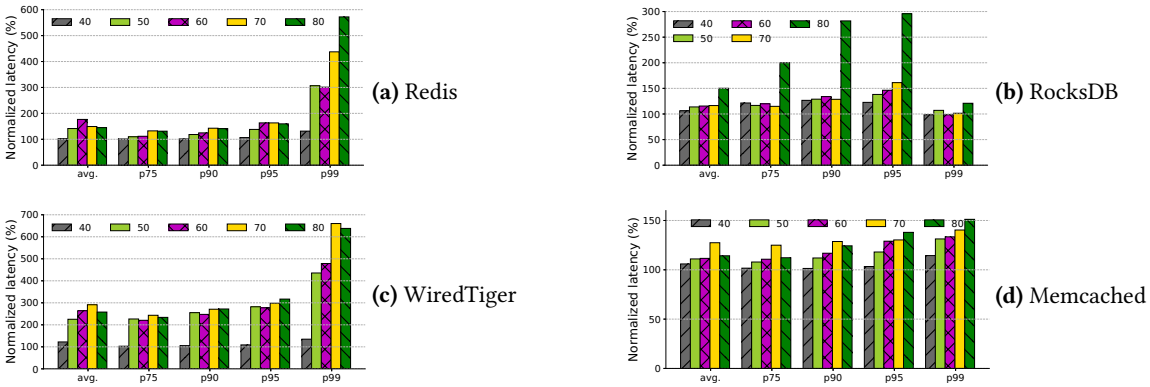


Figure 14: The normalized latency of four latency-critical services with different CPU suppression ratios S in Holmes.

a new set of HPEs via Instruction-Based Sampling [1] to diagnose memory controller congestion.

Thread invocation interval. The invocation interval of the monitoring and scheduling threads is a trade-off between overhead and query latency of latency-critical services. A longer interval incurs lower overhead while CPU adjustment may not be timely performed to achieve low latency, and vise versa. We suggest setting the invocation interval similar to the order of the query time of a latency-critical service. Thus, CPU adjustment can be timely performed for almost all queries. For example, the order of the query time in workload-a of Redis is around 50 to several hundreds of microseconds. The interval is set to 100μs. In fact, we set this interval for all four latency-critical services in the experiments. Note that users may want to examine this value to better fit for their production environments.

Where is Holmes in a system stack? Our prototype of Holmes runs as an individual daemon process in Linux user space. It can be integrated in cluster resource managers Yarn [48] and Kubernetes [6]. For example, Yarn's NodeManager is responsible for monitoring resource usage and adjusting core allocations and quotas.

7 RELATED WORK

Optimization for SMT. There are many efforts on optimizing application performance running on SMT servers [16, 18, 25–27, 34, 39, 40, 46, 51]. For example, three studies [18, 27, 39] dynamically adjust multiple resources, including memory bandwidth, CPUs and SMT in a job co-location environment. Heracles and Parties use feedback-based mechanism to conduct allocation for multiple resources. However, Bianchini *et al.* [14] and our study indicate that feedback-based mechanisms may render slow convergence to resource congestion. Caladan [27], a kernel-space approach, dynamically pauses/resumes threads of batch jobs running on siblings of a SMT core based on timeout from latency-critical services. Holmes differs from the three studies mainly in two aspects. It leverages CPU HPEs to quantify SMT interference on memory access, and it is a user-space approach that does not require modification to applications, libraries, or Linux OS. Holmes also finds that memory bandwidth is no longer a bottleneck in modern servers.

Latency reduction. Extensive efforts focus on reducing query latency for latency-critical services [13, 17, 22, 28–30, 32, 36, 38, 41, 50, 55]. For example, EvenDB [28] is a recent LSM Tree KV store that is optimized for data with spatial locality. FlatStore [22] is a recent KV store that uses logs in persistent memory for efficient data requests. There are also efforts on resource sharing and workload prioritization for latency-critical services. PerfIso [32] is an approach that uses native CPU isolation to achieve CPU sharing between latency-critical services and batch jobs. However, those studies do not consider job co-location in servers.

Resource sharing. Resource sharing and job co-location in multi-tenant systems have been extensively studied [12, 14, 15, 19–21, 35, 37, 42, 52]. For example, Mercury [35] is a hybrid resource scheduler that launches jobs with transient resources and kills jobs when the available resources drop below a threshold. BIG-C [19] is a preemption-based cluster scheduler that allows short jobs to preempt long jobs to achieve low latency and high utilization. However, those studies do not address SMT interference and its severe impact on performance of job co-location.

8 CONCLUSION

Holmes is a non-intrusive interference-aware CPU scheduler at user space for efficient job co-location in a SMT system. It tackles two challenges, 1) accurately diagnosing SMT interference on memory access by identifying hardware performance events and developing an quantitative method for interference measurement, and 2) adaptive CPU scheduling via interference-aware core allocation and CPU migration. Experiments show that Holmes achieves query latency of the latency-critical services close to that when the services are running alone in a server, while significantly improving server utilization and throughput of co-located batch jobs. Compared to PerfIso [32], Holmes reduces the average (99^{th} percentile) query latency by up to 49.0% (52.3%) for latency-critical services. It also significantly speeds up the convergence on resource allocation.

In the future, we plan to integrate Holmes with cluster management frameworks Yarn and Kubernetes.

9 ACKNOWLEDGMENT

This research was supported in part by U.S. NSF grant CCF-1816850.

REFERENCES

[1] Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf/.

[2] Introducing Hyperthreading into azure VMs. https://azure.microsoft.com/en-us/blog/introducing-the-new-dv3-and-ev3-vm-sizes//.

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

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

[5] Apache Hadoop. https://hadoop.apache.org/.

[6] Kubernetes. https://kubernetes.io//.

[7] LXC. https://linuxcontainers.org/lxc/.

[8] Memcached. https://memcached.org/.

[9] Redis. https://redis.io/.

[10] Rocksdb. https://rocksdb.org/.

[11] WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[12] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *Proc. of ACM HPDC*, 2016.

[13] D. S. Berger, B. Berg, T. Zhu, M. Harchol-balter, and S. Sen. Robinhood: Tail latency-aware caching – dynamically reallocating from cache-rich to cache-poor. In *Proc. of USENIX OSDI*, 2018.

[14] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, A.-M. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich. Toward ml-centric cloud platforms. *Commun. ACM*, 63(2):50–59, jan 2020. ISSN 0001-0782. URL https://doi.org/10.1145/3364684.

[15] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proc. of USENIX OSDI*, 2014.

[16] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proc. of USENIX ATC*, 2005.

[17] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. Hotring: A hotspot-aware in-memory key-value store. In *Proc. of USENIX FAST*, 2020.

[18] S. Chen, C. Delimitrou, and J. F. Martinez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proc. of ACM ASPLOS*, 2019.

[19] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of USENIX ATC*, 2017.

[20] W. Chen, A. Pi, S. Wang, and X. Zhou. Os-augmented oversubscription of opportunistic memory with a user-assisted oom killer. In *Proc. of ACM/IFIP Middleware*, 2019.

[21] W. Chen, A. Pi, S. Wang, and X. Zhou. Pufferfish: Container-driven elastic memory management for data-intensive applications. In *Proc. of ACM SoCC*, 2019.

[22] Y. Chen, L. Youyou, F. Yang, Q. Wang, Y. Wang, and J. Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of ACM ASPLOS*, 2020.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC*, 2010.

[24] J. Dean and L. A. Barroso. The tail at scale. *Communications of ACM*, 56(2): 74–80, Feb. 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. URL https://doi.org/10.1145/2408776.2408794.

[25] K. Deng, K. Ren, and J. Song. Symbiotic scheduling for virtual machines on smt processors. In *Proc. of IEEE CGC*, 2012.

[26] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-bandwidth aware thread allocation in multicore smt processors. In *Proc. of IEEE PACT*, 2013.

[27] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *Proc. of USENIX OSDI*, 2020.

[28] E. Gilad, E. Bortnikov, A. Braginsky, Y. Gottesman, E. Hillel, I. Keidar, N. Moscovici, and R. Shahout. Evendb: optimizing key-value storage for spatial locality. In *Proc. of ACM Eurosys*, 2020.

[29] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. King. Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones. In *Proc. of USENIX ATC*, 2018.

[30] M. Hao, H. Li, M. H. Tong, C. Pakha, and R. O. Suminto. Mittos: Supporting millisecond tail tolereance with fast rejecting slo-aware os interface. In *Proc. of ACM SOSP*, 2017.

[31] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE Data Engineering Workshops (ICDEW)*, 2010.

[32] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Proc. of USENIX ATC*, 2018.

[33] S. A. Javadi and A. Gandhi. Dial: Reducing tail latencies for cloud applications via dynamic interference-aware load balancing. In *Proc. of IEEE ICAC*, 2017.

[34] W. Jia, J. Shan, X. Shang, H. Cui, and X. Ding. vSMT-io: Improving i/o performance and efficiency on smt processors in virtualized clouds. In *Proc. of USENIX ATC*, 2020.

[35] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proc. of USENIX ATC*, 2015.

[36] M. Kogias and E. Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proc. of ACM Eurosys*, 2020.

[37] P. Lama, S. Wang, X. Zhou, and D. Cheng. Performance isolation of data-intensive scale-out applications in a multi-tenant cloud. In *Proc. of IEEE IPDPS*, 2018.

[38] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *Proc. of ACM PPoPP*, 2016.

[39] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proc. of ACM ISCA*, 2015.

[40] A. Margaritov, S. Gupta, R. González-Alberquilla, and B. Grot. Stretch: Balancing qos and throughput for colocated server workloads on smt cores. In *Proc. of IEEE HPCA*, 2019.

[41] P. A. Misra, M. F. Borge, I. Goiri, A. R. Lebeck, W. Zwaenepoel, and R. Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proc. of ACM EuroSys*, 2019.

[42] A. Pi, J. Zhao, S. Wang, and X. Zhou. Memory at your service: Fast memory allocation for latency-critical services. In *Proc. of ACM/IFIP Middleware*, 2021.

[43] M. Popov, A. Jimborean, and D. Black-Schaffer. Efficient thread/page/parallelism autoruning for NUMA systems. In *Proc. of ACM ICS*, 2019.

[44] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. In *Proc. of ACM VLDB Endowment*, 2016.

[45] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *Proc. of IEEE HPCA*, 2013.

[46] S. Siddha, V. Pallipadi, and A. Mallick. Chip multi processing aware linux kernel scheduler. In *Proc. of Linux Symposium*, 2005.

[47] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and M. Bernard. Unification of temporary storage in the nodekernel architecture. In *Proc. of USENIX ATC*, 2019.

[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.

[49] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *Proc. of USENIX NSDI*, 2020.

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

[51] X. Yang and S. M. Blackburn. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *Proc. of USENIX ATC*, 2016.

[52] Y. Yang, G.-W. Kim, W. W. Song, Y. Lee, A. Chung, Z. Qian, B. Cho, and B.-G. Chun. Pado: A data processing engine for harnessing ransient resources in datacenters. In *Proc. of ACM EuroSys*, 2017.

[53] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HOTCLOUD*, 2010.

[54] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-criticaltasks on shared multicore systems. In *Proc. of ACM ASPLOS*, 2016.

[55] T. Zhu, M. A. Kozuch, and M. Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providingtail latency slo guarantees. In *Proc. of ACM SoCC*, 2017.