CacheInspector: Reverse Engineering Cache Resources in Public Clouds

WEIJIA SONG and CHRISTINA DELIMITROU, Cornell University
ZHIMING SHEN, Exotanium Inc.
ROBBERT VAN RENESSE and HAKIM WEATHERSPOON, Cornell University
LOTFI BENMOHAMED and FREDERIC DE VAULX, National Institute of Standards and Technology
CHARIF MAHMOUDI, Siemens Corporate Technology

Infrastructure-as-a-Service cloud providers sell virtual machines that are only specified in terms of number of CPU cores, amount of memory, and I/O throughput. Performance-critical aspects such as cache sizes and memory latency are missing or reported in ways that make them hard to compare across cloud providers. It is difficult for users to adapt their application's behavior to the available resources. In this work, we aim to increase the visibility that cloud users have into shared resources on public clouds. Specifically, we present *CacheInspector*, a lightweight runtime that determines the performance and allocated capacity of shared caches on multi-tenant public clouds. We validate *CacheInspector*'s accuracy in a controlled environment, and use it to study the characteristics and variability of cache resources in the cloud, across time, instances, availability regions, and cloud providers. We show that *CacheInspector*'s output allows cloud users to tailor their application's behavior, including their output quality, to avoid suboptimal performance when resources are scarce.

CCS Concepts: • **General and reference** → **Measurement**; **Performance**; • **Computer systems organization** → **Cloud computing**;

Additional Key Words and Phrases: CPU cache, public cloud, cache latency, cache throughput

ACM Reference format:
Weijia Song, Christina Delimitrou, Zhiming Shen, Robbert van Renesse, Hakim Weatherspoon, Lotfi Benmohamed, Frederic de Vaulx, and Charif Mahmoudi. 2021. *CacheInspector*: Reverse Engineering Cache Resources in Public Clouds. *ACM Trans. Archit. Code Optim.* 18, 3, Article 35 (June 2021), 25 pages.
https://doi.org/10.1145/3457373

This work was partially supported by NSF grant #1955125, NIST grant #70NANB17H181, NSF CAREER Award CCF-1846046, and NSF grant NeTS CSR-1704742.

Authors' addresses: W. Song, 455 Gates Hall, Cornell University, Ithaca, NY 14853; email: wsong@cornell.edu; C. Delimitrou, 332 Rhodes Hall, Cornell University, Ithaca, NY 14853; enmail: delimitrou@cornell.edu; Z. Shen, 208 Cornell St, Ithaca, NY 14850; email: zshen@cs.cornell.edu; R. van Renesse, 433 Gates Hall, Cornell University, Ithaca, NY 14853; email: rvr@cs.cornell.edu; H. Weatherspoon, 427 Gates Hall, Cornell University, Ithaca, NY 14853; email: hweather@cs.cornell.edu; L. Benmohamed, NIST Headquarters, 100 Bureau Drive, Gaithersburg, MD 20899; email: lotfi.benmohamed@nist.gov; F. de Vaulx, NIST Headquarters, 100 Bureau Drive, Gaithersburg, MD 20899; email: frederic.devaulx@nist.gov; C. Mahmoudi, Siemens technology, 755 College Rd E, Princeton, NJ 08540; email: charif.mahmoudi@siemens.com.

1544-3566/2021/06-ART35
https://doi.org/10.1145/3457373

1 INTRODUCTION

Applications currently hosted on public clouds often experience unpredictable performance, which causes them to violate their **Quality-of-Service (QoS)** constraints [15, 24, 51, 52, 54, 65, 69]. There are many reasons behind performance unpredictability, including interference between applications sharing a server platform, unexpected spikes in user load, misconfigured resource reservations from the user's perspective, and system hiccups and failures [32, 42, 61, 62, 68]. Although public clouds provide some monitoring capabilities to users, these often do not extend to shared resources like the CPU cache, a significant culprit for contention and QoS violations.

The desire to reduce interference in shared caches has prompted a lot of recent work. Page coloring [41, 49, 70, 71], for example, is used by some cloud providers to avoid cache contention by removing sharing, albeit by reducing the available cache capacity per user. Similarly, the Cache Monitoring Technology (CMT) and **Cache Allocation Technology (CAT)** [12, 36] allow the OS or hypervisor to track cache usage, and dynamically partition cache capacity among co-scheduled applications [63, 68, 76].

Nonetheless, leveraging such techniques in the cloud remains challenging: (1) cloud providers have little visibility into the impact of cache allocations on end-to-end application performance; (2) several cloud infrastructures consist of older platforms, where these features are not available; and (3) cloud users themselves often do not have an accurate depiction of the cache resources allocated to their applications by the cloud provider [13, 29]. As we discuss in Section 5, the hints that the OS or hypervisor provide are often significantly over- or underestimated, resulting in unpredictable performance and QoS violations, especially for latency-critical, interactive services [10, 19, 20, 23, 24, 35, 42, 46, 47, 54, 57]. Notably, users have no visibility into how much cache is available to them at each point in time, which would allow them to adjust their application's behavior to better leverage available resources.

In this work, we bridge this gap between the visibility the cloud provider and the cloud user have into shared resources and allow cloud services to leverage this information to optimize their QoS. Specifically, we present *CacheInspector*, a lightweight CPU cache probing tool that offers cloud applications accurate, real-time information on the allocated cache resources of a multi-tenant public cloud. In particular, *CacheInspector* measures:

(1) The effective allocated capacity of each cache level;
(2) The throughput and latency of each cache level; and
(3) The throughput and latency of main memory.

CacheInspector can be used as either a stand-alone profiling tool or an application runtime disclosing cache resource dynamics. The application interface is simple and language independent. We have validated the ability of *CacheInspector* to determine the performance and capacity of each level in the memory hierarchy in a dedicated, controlled environment. We then use *CacheInspector* to quantify the variability and consistency of cache resources on public clouds, and find that cache allocations vary significantly over time, and across cloud providers, availability zones, and instance types. We also show a significant deviation between the cache resources that are advertised or reported by the OS (or hypervisor) and the effective cache capacity allocated to **virtual machines (VMs)**.

CacheInspector is compatible with all major **Infrastructure-as-a-Service (IAAS)** cloud providers, including Amazon EC2, Windows Azure, and Google Cloud Engine, and requires less than a second to measure cache capacity, allowing it to capture fine-grained changes in allocated

resources. Having this information allows cloud users to manage their available resources better. In Appendix A, we show how self-adjusting cloud services could benefit from *CacheInspector*. We open sourced *CacheInspector* [2] on GitHub for public access.

2 MOTIVATION

Cloud providers typically employ multi-tenancy to increase system utilization. Modern processors have deep cache hierarchies, whose size and sharing model varies across cache levels, with typical high-end server platforms like Intel Xeon sharing tens of megabytes of L3 cache among tens of cores. Cache sharing improves utilization, and—for cases where applications share working sets—it also improves performance. However, in the general case where applications with disjoint working sets compete for the limited shared cache capacity, cache sharing leads to contention and unpredictable performance. To avoid the negative effects of cache sharing, Intel platforms have introduced hardware and software support for cache partitioning (e.g., with techniques like Intel's CAT [21]). Configuring cache partitions in a way that preserves QoS requires a closer interaction than what is currently available between the cloud provider who has access to the partitioning techniques and the user who observes the impact of partitioning on application performance.

In the following, we show the impact of cache contention and cache partitioning on application performance using a set of popular scientific and deep learning applications:

- *pbzip2* is a multi-threaded version of *bzip2* [30]. We measure the time to compress the Linux kernel 4.13.4 source code.
- *mkl_mm* performs matrix multiplication with Intel's Math Kernel Library [38]. We measure the time for 1,000 $C \leftarrow A \times B$ operations, where A, B, and C are double float square matrices of order 1,000.
- *gcc* is the GNU C compiler. We measure the time needed to compile the Linux kernel 4.13.4.
- *mnist_deep* is a deep learning application in TensorFlow [13], which trains a CNN [45] to recognize handwritten digits. We measure the time for 500 iterations.

All applications' working sets in this experiment fit in main memory to avoid the impact of long disk I/O accesses. Additionally, we focus on L3 contention in an Intel CPU since it offers hardware support for partitioning; nevertheless, our approach can be applied to other CPU architectures.

2.1 Cache Contention

To simulate cache contention, we ran *cache poisoner* processes alongside the preceding applications. Each cache poisoner allocates and sequentially touches *int64_t* integers in a memory buffer with a stride equal to the cache line size. Using different buffer sizes, we can control the stress on the cache. Applications run on an Intel Xeon E5-2699 v4 processor with 22 CPU cores sharing a 55-MB L3 cache. We ran one application at a time on 8 cores and *cache poisoners* on the other 14.

Figure 1(a) shows the performance impact of L3 cache contention. The x-axis shows the buffer size used by the *cache poisoner*, and the right y-axis shows the L3 capacity available to the application, as measured by *CacheInspector*; the measurement approach is described in detail in Section 3. The left y-axis shows the application runtime, normalized to the execution time when running in isolation. All applications experience degraded performance as the *cache poisoner*'s footprint increases, with *pbzip2* taking 90% to 100% longer to complete when its available L3 capacity drops to 6 MB. *mkl_mm* is least affected by contention; nevertheless, its runtime still increases by 40% when available L3 capacity is 6 MB.

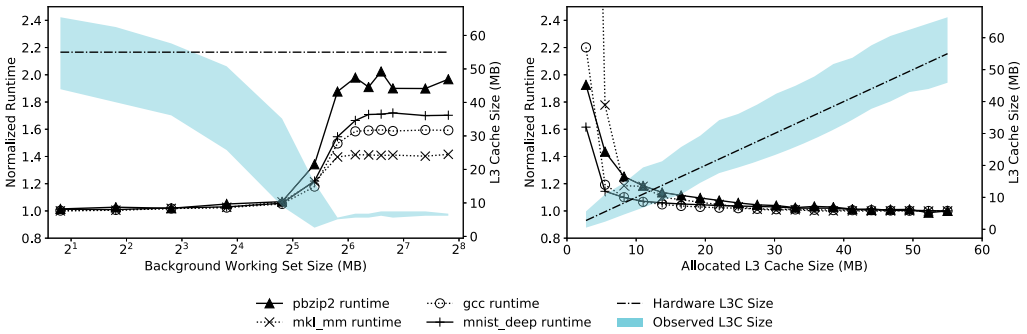


Fig. 1. Application performance with unmanaged cache contention (left) and with cache partitioning (right). The blue belts show the L3 cache range measured by *CacheInspector*.

2.2 Cache Partitioning

Cache partitioning allows cloud providers to isolate shared cache capacity across tenants. CAT [12] does so at cache way granularity. This avoids cache contention by allocating non-overlapping ways to each application but reduces the total cache size available per job.

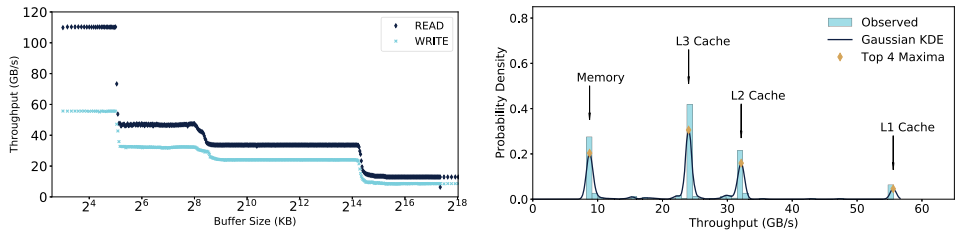
Figure 1(b) shows application performance as the allocated L3 partitions increase. The server's L3 cache has 20 ways with a total size of 55 MB. Each way corresponds to 2.75 MB of cache capacity. As before, applications run on eight cores, and L3 partitions change from one way to the entire 20 ways. *pbzip2*, *gcc*, and *mnist_deep* do not experience significant performance changes as long as the allocated L3 cache is greater than four ways (11 MB). Below four ways, they all experience dramatic performance degradations. *mkl_mm* is the most sensitive to L3 allocations, taking 20% more time to complete when allocated four cache ways, and 7× longer when allocated a single way, than when allocated the entire L3 (55 MB). Because CAT partitions cache capacity strictly, if the application working set is larger than the allocated L3 partition, the cache hit ratio can drop to the point where the benefit from having a cache at all is negligible [68]. Given the strong correlation between cache partitions and application performance, it is critical for cloud users to know how much cache capacity they are being allocated across virtual instances and cloud providers.

2.3 Application Tuning

In resource-constrained settings, the application often needs to be optimized to make the most out of the available cache capacity. In the event where its entire working set does not fit in the available cache, the user can prioritize the caching of the most performance-critical data. For example, Intel SSE/SSE2 provides non-temporal instructions that allow applications to control which data is cached [10, 47], and flushing everything else. Similarly, an application can force hot data to stay in the L3 using CAT. Once hot data is loaded into the allocated cache ways, the application calls CAT to unmap them so that the specific data cannot be evicted. Finally, if an application can tolerate some loss of output quality, as many machine learning and data mining applications can, it can sacrifice some precision in its output to improve execution time, under insufficient cache resources. For all of these optimizations to occur, an application needs to know precisely how much cache it is allocated at each point in time.

3 MEASUREMENT TECHNIQUES

CacheInspector is a lightweight runtime that measures the effective cache size available to users of a public cloud, as well as the throughput and latency of CPU caches and memory. *CacheInspector* includes a *profiling* and a *sampling* phase. In the profiling phase, *CacheInspector* deter-



(a) Throughput Cliffs (b) KDE

Fig. 2. Throughput exhibits cliffs as cache pressure increases (a) and using KDE to determine cache throughput (b).

mines the throughput and latency of each level of the memory hierarchy.[1] In the sampling phase, *CacheInspector* detects, for each cache level, the cache size available to the application at each point in time. The profiling phase involves introducing increasing pressure in each cache level through a set of carefully crafted microbenchmarks. This process can be time consuming, but only needs to happen once to determine the performance limits of each cache level and the memory. In contrast, the sampling phase only takes a few 100 ms to run and repeats periodically to capture changes in allocated cache capacity.

The methodology is platform independent and can be easily ported to other architectures with private and shared caches. The Intel Xeon servers we use have private L1 caches for instruction and data, private L2 caches, and a shared L3 cache. The code footprint of the examined applications is small enough to fit in the L1i caches, and hence we focus our analysis on data caches; however, the same methodology can be applied to instruction caches for larger cloud applications.

3.1 Profiling Cache/Memory Throughput

The memory system's throughput is largely affected by the cache capacity, application working set size, and access pattern locality. To quantify cache and memory throughput, we first measure the read/write throughput of each level in the memory hierarchy by varying the working set size. In general, as working set size increases throughput drops, often exhibiting cliff effects as larger, slower levels of the hierarchy come into play [16, 17, 25, 42, 67, 72]. Figure 2(a) shows such a stairstep graph for our server platform. Each stair level corresponds to the throughput of a particular cache level or memory. In the following, we describe how read throughput is measured; write throughput is measured similarly.

CacheInspector's throughput microbenchmark first allocates a memory buffer of a given working size, starting from a few kilobytes, and writes data in each entry ensuring that the memory pages are allocated, loaded, and cached. Then it repeatedly reads the buffer sequentially. For each read operation, the benchmark loads an aligned word to CPU registers. Depending on the architecture, the word can be 32-bit or 64-bit. We also recommend using SIMD instructions to stress the memory hierarchy. We leverage as many registers as possible, eight for Intel x86, to efficiently use the execution pipeline. Read throughput is computed based on the microbenchmark's total execution time. To avoid re-warming up caches and incurring unpredictable performance from the OS scheduler's actions, we disable the process scheduling using the real-time FIFO policy with the highest priority and pin the benchmark to a core.

[1]To avoid confusion, we note that profiling here considers available hardware resources rather than application performance.

ALGORITHM 1: Read Throughput Microbenchmark

```
1: Allocate memory buffer buf of size buf_sz
2: Fill buf with 0xa5
3: cnt ← num_traverse                                   ▷ register variable
4: Start Timer
5: while cnt > 0 do
6:     ofst ← 0                                         ▷ register variable
7:     while ofst < buf_sz do
8:         Load 1K bytes from buf+ofst to registers
9:         ofst ← ofst + 1024
10:    end while
11:    cnt ← cnt − 1
12: end while
13: End Timer
14: t ← ReadTimer()
15: return buf_sz × num_traverse/t
```

Algorithm 1 shows the pseudocode of the read throughput microbenchmark. `buf_sz` specifies the size of the buffer to read from. `num_traverse` specifies how many times the buffer is traversed. `clock_gettime`[3] can take tens of nanoseconds, which are not negligible when requiring precise throughput measurements. Therefore, the value of `num_traverse` is chosen large enough to amortize the overhead of clock time reading and small enough to be time efficient. We set the buffer size to 128 MB, which completes in 10 ms.

Line 8 moves 1,024 bytes as a batch to amortize the overhead of loop execution, which includes two instructions—one to increment the counter and another for the conditional jump—a much more expensive implementation despite the branch predictor's effectiveness for the specific loop. In doing so, it only adds 0.016 instructions to move a 64-bit value. Using even larger batches is possible; however, it runs the risk of thrashing the instruction cache. We also leverage single instruction multiple data (SIMD) parallelism like AVX [53] and Neon [64] to intensify the workload so that the performance difference between cache levels is clear.

By changing the buffer size, we can measure the performance of each cache level and of memory. Figure 2(b) shows the density of the data points in Figure 2(a) grouped by throughput. Using **Kernel Density Estimation (KDE)** with a Gaussian kernel, we extract the maxima—marked by brown dots—which correspond to the height of the plateaus in Figure 2(a), and capture the throughput of the different cache levels and main memory.

To obtain the KDE curve, we start with one-fourth of the reported L1 cache size and increase it by 2% at a time, until it reaches 256 MB. This procedure results in approximately 500 buffer sizes, sufficient to cover all cache levels. Although a wider buffer size range is possible, given typical L3 sizes in modern servers, it would significantly increase the profiling time. From the KDE curve above, we calculate the top N maxima, where N is the number of cache levels plus one for main memory.

Discrepancies with OS statistics. The OS usually reports cache information including the number of levels and sizes (e.g., through `sysfs` in Linux). This information is either read directly from hardware or provided by the hypervisor; however, it does not always accurately capture the amount of resources available to a process (see Section 5). Even though *CacheInspector* does not rely on the OS-reported cache sizes, KDE requires knowing the correct number of cache levels to extract maxima. We use density-based spatial clustering (DBSCAN) [28] to verify the number of cache levels reported by the OS. DBSCAN's ϵ parameter determines the density threshold in the KDE

curve. We empirically set ϵ to 0.03GBps, which correctly identifies the number of cache levels in all of our experiments.

Although cliff effects have been long observed in cache hierarchies, we found that the drop in throughput is not always steep. If the cache replacement policy were simple LRU, the performance drop ought to be sudden (*cliff effect* [17]), because a sequential workload would not benefit almost at all from a cache that is not large enough to entirely fit the buffer. However, modern CPUs employ more sophisticated replacement policies than LRU, which allow the microbenchmark to benefit from caching, despite its streaming nature [40, 63, 75].

3.2 Profiling Cache/Memory Latency

A common way to test memory latency is to use a randomized circular singly linked list [26]. While traversing the linked list, the CPU needs to wait until the address has been loaded into a register by the current read instruction before it can begin executing the next. By counting how many reads are performed in a given period of time, we can calculate the average latency of a memory access.

ALGORITHM 2: Latency Microbenchmark

```
1:  Allocate memory buffer buf of size buf_sz
2:  Initialize a randomized circular linked list in buf
3:  cnt ← num_read                                   ▷ register variable
4:  ent ← 0                                          ▷ register variable
5:  Start Timer
6:  while cnt > 0 do
7:      ent ← buf[ent]                               ▷ 128 times
8:      ...
9:      ent ← buf[ent]
10:     cnt ← cnt − 128
11: end while
12: End Timer
13: t ← ReadTimer()
14: return t/cnt
```

Algorithm 2 shows the pseudocode for the latency microbenchmark in *CacheInspector*. As with the throughput microbenchmark, to amortize the loop management overhead, we perform 128 read instructions in each loop iteration. buf_sz is the memory size of the linked list. num_read is the number of read operations we issue. We set $num\_read = 2^{20}$ so that each round takes several milliseconds, dwarfing the overhead of reading the timer. As before, we disable the scheduler and pin the microbenchmark to a core.

We model the average memory access latency in Equation (1), where L_m is the measured average latency, n is the number of cache levels, H_i is the hit ratio in the i^{th} cache/memory level, and L_i is the latency of the i^{th} cache/memory level we want to determine. We use level $n + 1$ to represent memory.

$$L_m = \sum_{1}^{n+1} H_i * L_i \tag{1}$$

If hardware performance counters are available, it should be easy to measure the cache hit/miss counts [12, 29, 73]. Based on observed latencies and corresponding hit counts, we can determine cache latencies by linear regression. However, in a public cloud, such resources are usually hidden from a cloud user. Therefore, we design a hardware-independent approach that works excitingly well.

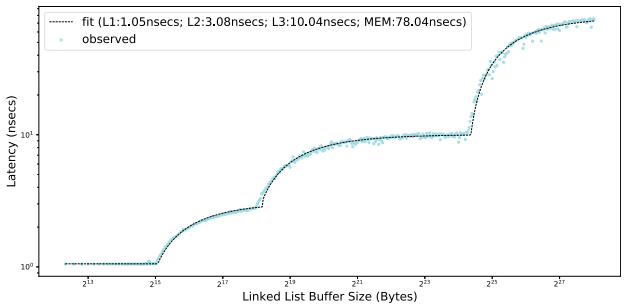


Fig. 3. Cache latency measurement and curve fit using the latency model derived from Equations (1) and (2).

In Equation (2), we model the hit rate, which is essentially the probability of a memory access being served by a specific cache level. The independent variable x represents the size of the linked list buffer. The parameter S_i is the size of the i^{th} cache level. We define $S_0 = 0$ and $S_{n+1} = x$ for generality.

$$H_i(x) = \begin{cases} 0 & \text{if } x \le S_{i-1} \\ \frac{x - S_{i-1}}{x} & \text{if } S_{i-1} < x \le S_i \\ \frac{S_i - S_{i-1}}{x} & \text{if } x > S_i \end{cases} \tag{2}$$

The first condition means the size of the linked list buffer is smaller than the $(i-1)^{th}$ cache level. Ideally, the buffer should be well fit in the $(i-1)^{th}$ cache level without accessing the i^{th} cache level. The second condition means the linked list buffer is greater than the $(i-1)^{th}$ cache level but smaller than the i^{th} cache level. Ideally, the buffer will use the whole $(i-1)^{th}$ cache level, and only $(x - S_{i-1})$ of the buffer will be cached at level i. Since the linked list access is unifiedly randomized, $H_i(x)$ is proportional to the size of data cached in the i^{th} level, but not higher. The third condition means the linked list buffer is greater than the i^{th} cache level. It is easy to deduce $H_i(x)$ similarly.

Combining Equation (1) and Equation (2), we can get a function $L_m(x)$ with $2n+1$ parameters: $S_1, S_2, \ldots, S_n$ and $L_1, L_2, \ldots, L_{n+1}$. By running Algorithm 2 with the linked list buffers of different size, we can collect many $(x, L_m(x))$ data points. Please note that $L_m(x)$ is a continuous function, allowing us to fit function $L_m(x)$ with non-linear least square to determine those parameters. Figure 3 shows the measurement of latency (blue dots) along with the fitted model (dashed line). We use the same experiment environment and buffer size set in Figure 2(a). It is clear that the fitted model matches the observed value very well.

Obtaining the actual cache/memory latency is more complicated than Equation (1) implies. For instance, the L1 latency depends on the addressing mode [6, 7], and the L3 latency varies depending on the cache coherency operations involved [12]. In some processors, like the Intel Core microarchitecture, cache latency is different for load and store instructions. For the sake of simplicity, we did not separately measure load and store latencies in this work.

3.2.1 Running in Virtualized Environments. **Translation Lookaside Buffer (TLB)** misses are common during the latency profiling phase because memory accesses are randomly distributed across tens of megabytes, increasing TLB contention. Worse, the TLB miss penalty is much higher in hardware-assisted virtualization environments due to nested page tables [14, 18]. In extreme situations, a TLB miss inside a guest OS could use five times more memory references than one outside [18]. Since our approach does not exclude this penalty, the measured latency would be significantly higher than the real value. Our test in a KVM guest with an Intel Sandy Bridge CPU shows that the measured L3 cache latency is twice that of ground truth.

To reduce the number of TLB misses, we enable huge pages [74]. x86 CPUs normally supports 2-MB and 1-GB huge pages capable of caching the translation entries for 64-MB and 4-GB data, respectively, in an L1 TLB. Although the largest page size eliminates most of the TLB misses, in our experiments we did not observe measurable differences between 2-MB and 1-GB huge pages. We believe enforcing huge page table is feasible because cloud providers usually make it available to cloud users [5, 9].

3.2.2 Cache Associativity. Most of the modern CPU cache architectures use set-associative policy. With a fixed size, the more ways a cache has, the more cache slots data can use, but the more expensive it becomes. CPU architectures adopt different set/way layouts for different trade-offs between performance and cost.

For example, Intel Skylake and Intel Sandy Bridge CPUs both have 256-KB L2 cache per core. The former's is four-way 1,024 sets, and the latter's is eight-way 512 sets. The cacheline size is 64 bytes. Consider a 1-MB randomized circular linked list in contiguous physical memory. In Skylake, 1,024 bytes (size of 16 cache lines) of the list share a cache set of 4 cache lineswhereas in Sandy Bridge, 2,048 bytes (size of 32 cache lines) share a cache set of 8 cache lines. Although a Sandy Bridge core has more eviction choices, it does not get better performance than a Skylake core in Algorithm 2 because all eviction candidates have the same probability of being accessed in the future, due to the randomness of the data. Therefore, cache associativity does not affect latency profiling.

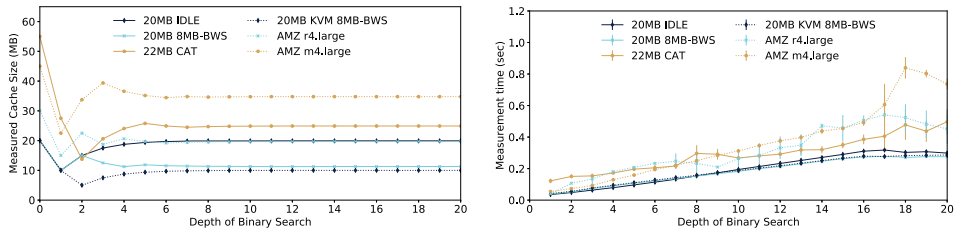
Please note that the paging mechanism in OSs may break the assumption that data evenly shares the cache sets. Uneven sharing will change the probability distribution we used in Equation (2). To avoid this uneven sharing, we again enforce huge pages so that a single page can cover all available cache sets, satisfying the evenness assumption.

3.3 Sampling Phase

In the sampling phase, we determine the available capacity in each level of the cache hierarchy. Since the sampling phase is repeated frequently to capture changes in cache allocations, it needs to be lightweight. We achieve this by employing binary search in the results of Figure 2(a) to determine available cache capacities. The x coordinates of the cliffs in Figure 2(a) correspond to cache sizes. Additionally, the y coordinates of the plateaus are known from throughput profiling. Based on this, we determine the x coordinate of a cliff using Algorithm 3.

Algorithm 3 has three input parameters. T is the cache/memory throughput performance obtained with throughput profiling, in descending order. We can measure either read or write throughput. H contains the cache sizes reported by the OS, ordered by their distance from the core. D is the depth of the binary search. For each cliff between adjacent entries in T, we search for the working set size that corresponds to a throughput halfway from the top to the bottom of the cliff. Line 12 calls Algorithm 1 to determine the throughput at buffer size h. Figure 2(a) shows that increasing the buffer size may occasionally result in a slightly higher throughput, presumably due to the efficacy of the cache replacement policy. Therefore, throughput is not a monotonic function over the buffer size, which can theoretically introduce inaccuracies in the above estimation since binary search assumes a sorted input. In Section 4, we show that this estimation is accurate in practice.

The depth of the binary search D controls the overhead and accuracy of *CacheInspector*. Too few runs would lead to inaccurate cache size estimates, whereas too many runs would incur high measurement overheads. Figure 4(a) shows the convergence of the measured cache size as the binary search depth increases in four controlled private environments and in two public cloud instances. We tested *CacheInspector* on an Intel Xeon E5-2690 v0 processor with a 20-MB L3 cache,



(a) Binary search exploration (b) Measurement overhead

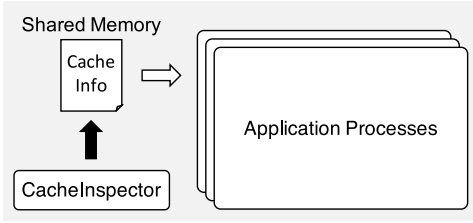
Fig. 4. Selection of the binary search depth.

ALGORITHM 3: Computing Cache Sizes

```
1:  cache_sizes ← []
2:  for i ← 1, len(T) − 1 do
3:      Y ← (T[i] + T[i + 1])/2                          ▷ target throughput
4:      cache_sizes[i] ← BSEARCH(Y, H[i], 0, 0, D)
5:  end for
6:  return cache_sizes
7:
8:  function BSEARCH(y, h, l, u, d)
9:      lb ← l                                           ▷ lower bound
10:     ub ← u                                           ▷ upper bound
11:     nh ← h                                           ▷ size hint
12:     thp ← throughput at h
13:     if thp < y&&d > 0 then
14:         lb ← h
15:     else if thp > y&&d > 0 then
16:         ub ← h
17:     else
18:         return h
19:     end if
20:     if ub = 0 then
21:         nh ← nh × 2
22:     else
23:         nh ← (lb + ub)/2
24:     end if
25:     return BSEARCH(y, nh, lb, ub, d − 1)
26: end function
```

without any background workload (`20MB IDLE`). In the same environment, we started a process touching 8-MB memory repeatedly to simulate a moderately memory-intensive background task (`20MB 8MB-BWS`). We used the same server but ran *CacheInspector* in a KVM guest OS (`20MB KVM 8MB-BWS`), and we additionally tested *CacheInspector* on an Intel Xeon E5-2688 v4 (55-MB L3) using CAT to allocate a 22-MB L3 partition to the application (`22MB CAT`). Finally, we ran *CacheInspector* on Amazon EC2 with `r4.large` and `m4.large` instances (`AMZ r4.large` and `AMZ m4.large`).

We see that binary search converges after eight to nine rounds. For L1 and L2, the measured size converges even faster because the search range is tighter. Figure 4(b) shows the measurement time versus the binary search depth. The total time required for binary search to converge is 240 to 300 ms.



(a) Architecture

```
#define MAX_CL          (4)
#define MAX_DS          (16)
typedef struct {
    union {
        uint64_t    cache_size[MAX_CL][MAX_DS];
        uint8_t     _bytes[PAGE_SZ];
    } page;
} cache_info_t;
extern const cache_info_t* get_cache_info();
```

(b) API

Fig. 5. *CacheInspector* runtime.

In some cases, the cliffs of Figure 2(a) can be wide, which has an impact on application performance. To account for such cases, we examine multiple working set sizes within a cliff's range. For example, if the throughput of L3 cache and memory are 17 GBps and 10 GBps, respectively, *CacheInspector* will examine working set sizes for throughputs of 16 GBps, 15 GBps, . . . , down to 11 GBps.

3.4 *CacheInspector* Runtime

We implemented a runtime to disclose the dynamic cache information collected during the sampling phase, as shown in Figure 5(a). The core of this runtime is a data structure called `cache info`, which is a table containing the cache size measurements. The size of `cache info` is exactly one page to fit in a single page in shared memory. Application code reads `cache info` using the only `get_cache_info()` API defined in the *CacheInspector* library as shown in Figure 5(b). Based on the cache information, an application can adapt itself for performance, for example, using the Protean code mechanism [47].

To run an application, *CacheInspector* runtime first initializes the `cache info` in shared memory and then launches the application process(es). The runtime periodically pauses the application processes and executes a sampling phase to collect the cache size information in `cache info`. Higher sampling frequency captures more dynamics but introduces higher overhead. The runtime flexibly allows user control over the sampling frequency for different demands. The user can also limit the sampling on a specific cache level to lower the overhead because, usually, only the last-level cache is shared and dynamic.

3.5 Other Implementation Considerations

3.5.1 Power Management. Modern OSs leverage dynamic voltage and frequency scaling to balance between CPU power consumption and performance. In addition, new processors overclock to boost CPU performance when load is high [4, 59]. Both dynamic voltage and frequency scaling and overclocking take microseconds to take effect. To ensure that our benchmark runs in a consistent environment, we introduce a warming up stage, which runs the benchmark for several rounds but ignores results.

New power management hardware smartly controls the frequency of each CPU core separately according to power policy as well as thermal constraints. For instance, the Intel Xeon E5-2699 v4 processor allows only 2 out of 22 cores to overclock to a maximum of 3.6 GHz simultaneously. If all 22 cores are busy, they can overclock to 2.8 GHz simultaneously. Although some cloud providers allow control over CPU frequency from inside specific instances (e.g., Amazon `m4.10xlarge`), cloud users cannot directly control CPU frequency in most cases. Thus, the profiling stage and sampling

stages may run at different CPU frequencies, resulting in inaccurate cache size measurements. Therefore, we measure the L1 cache latency in the beginning of the sampling phase. If this L1 cache latency does not match what we measured during profiling, we drop the sampling data.

3.5.2 Cache Inclusion Policy. In a multi-level cache architecture, if a cache level must contain the data in a higher level, it is called *inclusive cache*. On the contrary, if a cache level must not contains the data in a higher level, it is called *exclusive cache*. A third policy is called *non-inclusive non-exclusive*, which is neither strictly inclusive nor strictly exclusive [22]. Although inclusive cache policy are common, some processors adopt the non-inclusive non-exclusive policy to avoid back-invalidation [39]. *CacheInspector* currently assumes inclusive policy for all cache. When non-inclusive cache is involved, the sampling stage might over-report the size of non-inclusive cache level if it is shared, because the data cached in a higher level will not be evicted when the corresponding entries are evicted from non-inclusive cache by a competing core. However, as sequential workload minimizes the number of useful entries in the higher cache level(s), we cautiously neglect the preceding effect of non-inclusive cache policy. We plan to investigate the non-inclusive policy more thoroughly in future work.

3.5.3 Multicore Instances. To evaluate a VM with multiple cores, we pin the microbenchmark on one core and leave the other cores idle. The public cloud hypervisor may share one physical core among the VMs, and users have no control over the affinity between virtual and physical cores. In this case, our benchmark competes for compute and private cache resources and reports the allocated shares as shown in Section 2.1. Sharing physical cores only occurs for the smallest offered instances and is unlikely in practice.

3.5.4 Security. Security is a vital issue in the cloud. We argue that *CacheInspector* does not introduce vulnerabilities into the existing system. *CacheInspector* runs without application-specific information or exposing application information. The coarse grind throughput and latency measurement data collected from *CacheInspector* is hard to be useful for side-channel attacks like Meltdown [50], Spectre [44], and Flush+Reload [77]. Please note that Meltdown and Spectre require the latency data of separate cache lines, whereas *CacheInspector* can only provide the latency estimation of a whole cache level. *CacheInspector* does detect the time pattern of available cache resources in a shared environment, as shown in Section 5. However, it is easy for a cloud provider to isolate sensitive applications with CAT [12] when such a protection is required. We claim that using *CacheInspector* is orthogonal to cloud security.

4 *CACHEINSPECTOR* VALIDATION

We first evaluate *CacheInspector*'s accuracy of detecting cache sizes, as well as latency using four different private servers. We use private servers for the validation study, avoiding the uncontrollable interference from external workloads on a public cloud. Next we evaluate *CacheInspector* using a variety of VM instances in both public and private clouds. Due to the time and cost involved, we have not tested *CacheInspector* on all of the hundreds of types of instances available on public cloud providers. Instead, we selected 10 types of instances from Amazon North Virginia, Google US-Central, Microsoft Azure US East US2, and a private cloud deployment called *Fractus*. Then we use trace-driven experiments to demonstrate how cache information can benefit applications.

We measure cache size on two private servers with different hardware and configuration settings to ensure *CacheInspector*'s generality and platform independence. Server 1 has two Intel Xeon E5-2690 v0 processors and 96 GB of memory. Each of its processors has 8 cores with 32 KB of L1 data cache and 256 KB of unified L2 cache per core, and shared 20 MB of unified L3 cache. Server 2 has two Intel Xeon E5-2699 v4 processors and 128 GB of memory. Each of its processors has 55 MB of

Table 1. Cache Size Measurement Accuracy

Server	Level	Documented (KB)	Measured (KB)	
			Bare Metal	Virtualized (KVM)
Server 1	L1D	32	32.4 (0.49)	32.5 (0.5)
	L2	256	312.8 (32.6)	294.8 (53.4)
	L3	20,480	19,982 (70.0)	18,703.3 (873)
Server 2	L1D	32	34 (0)	31.9 (1.66)
	L2	256	265 (15.1)	223.8 (31.1)
	L3	56,320	50,981.8 (681)	44,393 (708.6)
Server 2 + CAT	L1D	32	32.75 (1.09)	32.6 (0.8)
	L2	256	227.25 (42.5)	236.6 (17.0)
	L3	22,528	19,950.8 (1189)	16,494 (721.8)

Table 2. Latency Measurement Accuracy

Level	Documented (ns)	Measured (ns)			
		Regression	Bare Metal (2M page)	KVM (2M page)	KVM (4K page)
L1D	1.05	1.06	1.05	1.06	1.06
L2	3.16	3.17	3.08	3.24	3.45
L3	10.00~11.32	10.65	10.04	10.49	25.12
Mem	N/A	75.96	78.04	106.11	115.17

L3 cache shared among 22 cores with CAT capability, and its per-core L1 and L2 cache sizes are identical to those in Server 1.

The servers run Linux 4.x and KVM hypervisor. The KVM guests also run Linux 4.x. We run *CacheInspector* in the host OS (Bare Metal) as well as in a KVM virtual machine (KVM). We pin the *CacheInspector* process to one CPU core to avoid being scheduled on different cores. In CAT experiments, we use the `rdtset` command to allocate 8 of the 20 cache ways to the *CacheInspector* process. If the *CacheInspector* is running inside KVM guest, we allocate the same amount of cache ways to the KVM process since CAT is not available inside the VM. Table 1 compares the measured cache size with ground truth from hardware documents. The measured cache sizes are the centers of corresponding cliffs found by Algorithm 3. We run it five times and calculate the average—the numbers in parentheses report standard deviation. We did not run any background workload during the measurement.

We can see that L1 cache measurements are within 6% of the documented values. L2 and L3 cache measurements are also close to the documented values, but more scattered than L1 data because, as shown in Figure 2(a), the performance cliffs falling from L2 and L3 are less steep. The cache replacement policies for L2 and L3 are likely different from L1's, but we have not yet found details in publicly available information. Unlike the L1 data cache, L2 and L3 caches hold both data and code.

Table 2 shows cache latency profiling on Server 1. The documented latencies are calculated according to Intel's manual [11]. The `Regression` column derives from the cache hit ratio, which is measured by hardware performance counter, using linear regression and the model in Equation (1). Please note that the regression approach does not involve Equation (2), because the hit rate is known. The rest of the numbers are determined by Algorithm 2. By default, we use a 2-MB huge page to minimize the TLB miss penalty. We also show cache latencies measured inside a KVM guest

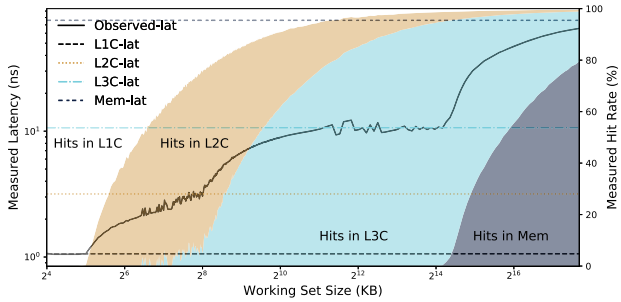


Fig. 6. Determining cache latency and hit ratio. The observed latency rises dramatically when the working set size grows beyond the cache boundary. The stacked hit ratio shows the percentage of the cache level in which memory accesses are ultimately served.

with normal 4-KB pages. The regression results are close to the documented values. Although some cloud providers do expose program counters to cloud users in very limited types of instances,[2] most cloud providers disable them for security and complexity reasons. Fortunately, the results from Algorithm 2 have only slightly higher errors than the regression results if 2-MB huge pages are used. Without a huge page, Algorithm 2 will incur a much higher overhead than with a huge page.

Figure 6 shows the relationship between measured latency and cache hit ratio. The stacked area shows the hit ratio for each cache level. Each time the working set size grows across the L_n cache boundary, the observed latency grows abruptly because the L_n cache hit ratio drops quickly, as shown at the top of Figure 6. This effect, however, levels off when the working set size grows further and the hit ratio drops correspondingly. When the working set size grows close to the size the of L_{n+1} cache, the latency begins to jitter because the working set begins to overflow out of the L_{n+1} cache and into the next level. The dashed horizontal lines correspond to the regression latencies in Table 2.

To show that our approach is architecture independent, we also validated *CacheInspector* on CloudLab *m400* servers with X-Gene APM883208-X1 processors [26]. Please refer to the *CacheInspector* GitHub repository [2] for the experiment result.

4.1 *CacheInspector* Overhead

The *CacheInspector* profiling phase involves running Algorithms 1 and 2 approximately 500 times with different buffer sizes. Initializing random cyclic linked lists with tens of millions of entries in latency profiling (Algorithm 2) is time consuming. It took more than 4 minutes to get a data point in Figure 3 when the buffer size is 100 MB. Fortunately, the profiling phase runs only once at the beginning. Therefore, we treat it as deployment overhead. We focus on the sampling overhead in *CacheInspector* runtime in the rest of this section.

We evaluated the runtime overhead using the *pbzip2* application described in Section 2. We first run *pbzip2* without enabling *CacheInspector* sampling and use its running time as the baseline. Then, we run the same workload with a sampling phase triggered once every 2, 5, 10, 20, and 40 seconds. In the sampling phase, we limited *CacheInspector* to evaluate only the shared last-level cache allocation. We keep the unused cores idle to avoid interfering with *pbzip2* execution. We run each test five times and calculate the average and standard deviation. Figure 7 shows the results, clustered in the number of threads. It is clear that the higher the sampling frequency is, the more

[2] Amazon EC2 exposes the program counters in dedicated instances like `m4.16xlarge` and `i3.16xlarge` [34].

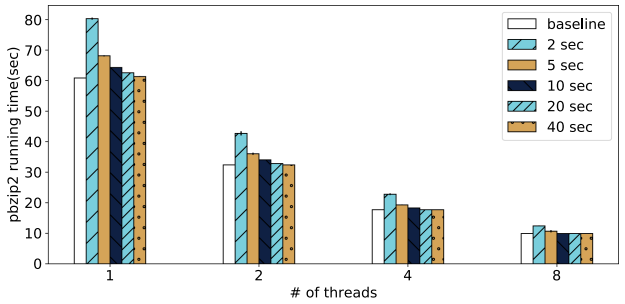


Fig. 7. *CacheInspector* runtime overhead.

overhead it incurs. With a 2-second interval, *CacheInspector* adds about 20% to the baseline running time. When the frequency drops to once every 20 seconds, the overhead is at most 2.5%. It translates to 400- to 500-ms running time for each sampling phase, including the overhead of stopping and resuming the application process(es). The number matches our observation in Figure 4(b). This trend is not sensitive to the number of threads involved in this application because the overhead of stopping and resuming threads is negligible compared to the sampling overhead. The almost invisible error bars show that the overhead is consistent across each run.

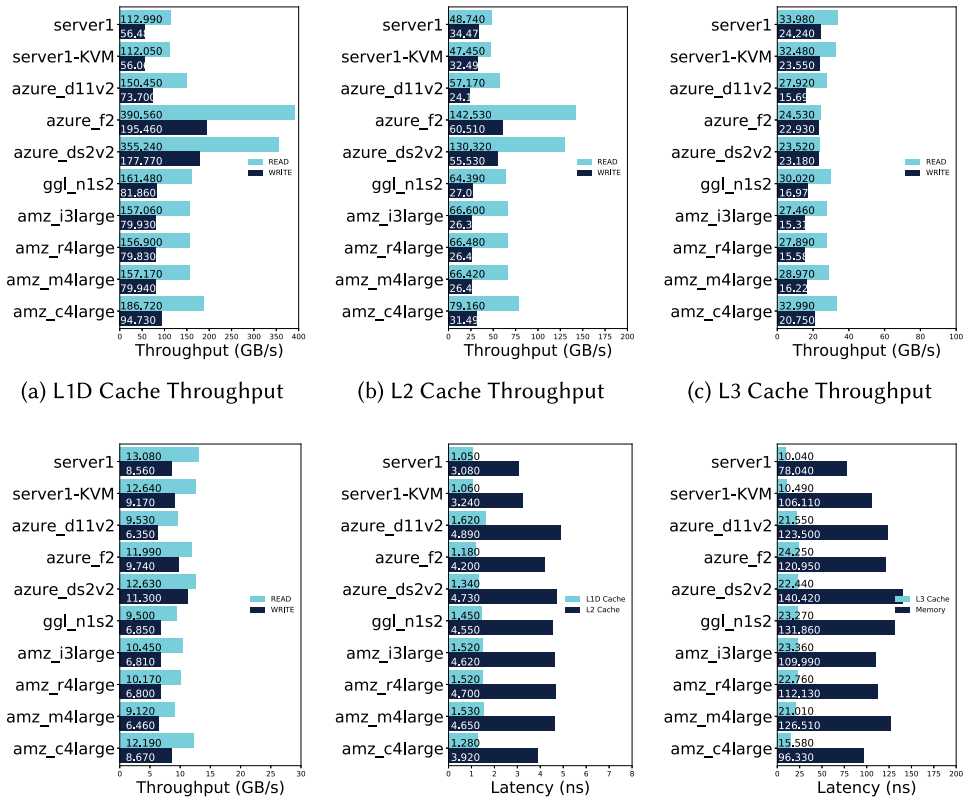
5 CACHE RESOURCES IN PUBLIC CLOUDS

We measured the cache resources in different types of VM instances from multiple cloud providers. Due to the time and cost involved, we have not tested *CacheInspector* on all of the hundreds of types of instances available on public cloud providers. Instead, we selected 10 types of instances from Amazon North Virginia, Google US-West, Microsoft Azure US East US2, and a controlled local environment to show the profiling results. An important consideration is the stationarity of these measurements. To this end, we monitored cache size measurement fluctuation in days using four instances from Amazon, Google, and Azure.

5.1 Throughput and Latency Profiling

Figure 8 compares throughput and latency profiling of CPU caches and memory for various VM instance types. The first two rows are for the bare metal and virtualized instances running on a local server, Server 1, as we used in Section 4. The others are for cloud instances.

From the results in the controlled environment, we can see that virtualization adds negligible overhead to throughput and latency of the memory hierarchy. Figure 8(a) and (c) show that `azure_f2` and `azure_ds2v2` instances have much higher L1 and L2 cache throughput than the others. That is because they use a newer CPU model with AVX512 support, allowing processing 512-bit data per instruction. The others only have AVX2 capable of processing 256-bit data per instruction. We noticed that the measured performance of the L1/2 cache of local servers is significantly higher than that in the cloud despite the hardware difference. According to the L1 cache latencies, *CacheInspector* can reach the highest Turbo Boost frequency, whereas cloud instances do not. We monitored a couple of cloud instances in Amazon and Google for weeks but found no evidence that they ever boosted to the highest CPU frequency. For example, the Amazon EC2 `m4.large` instance has an Intel E5-2686 v4 core as reported, which can overclock to 3 GHz. However, our test results show that its L1 cache latency is 1.53 ns. Since each L1 access uses four instructions, we can infer the CPU frequency is around 2.6 GHz. The hardware manual points out that each core can boost to at most 2.7 GHz if all of them boost simultaneously. The coincidence



(a) L1D Cache Throughput (b) L2 Cache Throughput (c) L3 Cache Throughput

(d) Memory Throughput (e) L1D and L2 Cache Latency (f) L3 Cache and Memory Latency

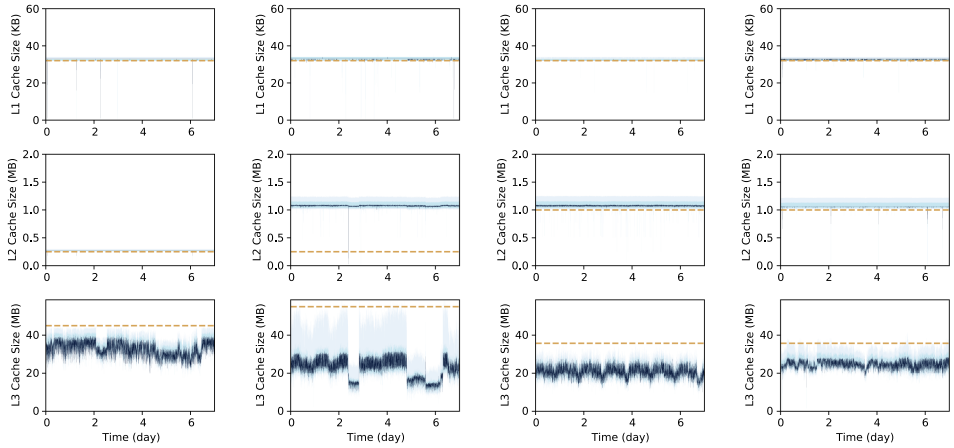
Fig. 8. Cache and memory performance in a controlled environment and three public cloud providers.

implies that the cloud provider might throttle the CPU frequency for fairness in the cloud. But it could also be the case that all other cores are busy. We have not validated our speculation with cloud providers yet.

5.2 Stationarity

To understand how effective cache sizes change over time, we ran *CacheInspector* sampling every minute in Amazon EC2, Google Cloud Platform, and Microsoft Azure Cloud for one week. Due to our limited budget, we selected four instances as shown in Figure 9. A dark blue line shows the time series of estimated cache size. Recall that Algorithm 3 determines the cache size using a buffer size corresponding to the throughput halfway from the top to the bottom of a cliff (Figure 2(a)). To estimate the width of a cliff, we pick five extra buffer sizes, two larger and two smaller than the estimated cache size, whose throughput measurement increase (decrease) evenly to the faster (slower) level in the cache/memory hierarchy, as we illustrate in the last paragraph in Section 3.3. The light blue scaled belts in Figure 9 show the distances between those buffer sizes. The brown dashed lines show the cache sizes reported by the OS.

We can see that in Amazon EC2 (Figure 9(a)) and Azure Cloud (Figure 9(c) and (d)), the L1 and L2 cache measurements agree with OS-reported sizes. The L3 cache size measurements fluctuate significantly and are much smaller than the stated size because our instances share the L3 cache with others. This is because the L3 cache is shared and co-located instances have a fluctuating



(a) Amazon `m4.large` (b) Google `n1s2` (c) Azure `ds2v2` (d) Azure `f2`

Fig. 9. Cache size stationarity in the public cloud. The belts show how the cache range measured with Algorithm 3 changes over 3 days. Dashed lines show the cache sizes reported by the OS. We randomly pick the instances from cloud providers as follows: an `m4.large`(9(a)) instance from Amazon AWS `us-east-1` region, an `n1-standard-2`(9(b)) instance from Google `us-west1c` region, a `ds2v2`(9(c)) instance from Azure `east-2` region, and an `f1`(9(d)) instance from Azure `east-2` region.

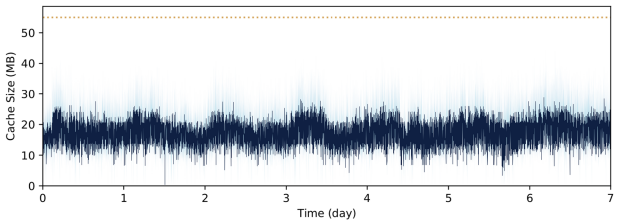
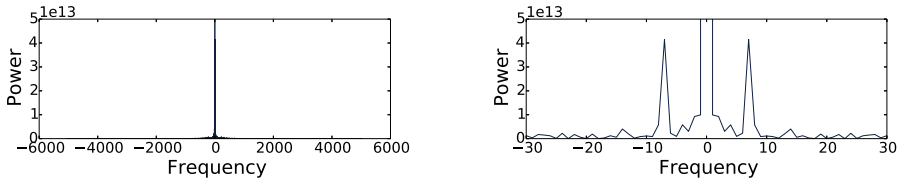


Fig. 10. An example of diurnal cycles of L3 cache measurements.

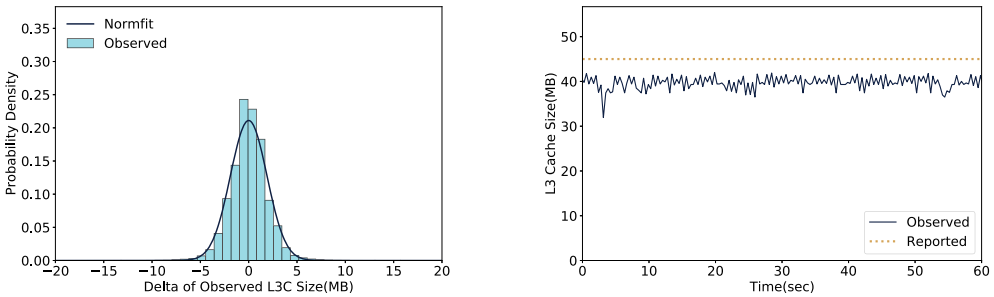
workload. The L3 cache size measurement in Figure 9(b) shows a cliff pattern: the observed measurements are stable for hours but suddenly drop or rise by more than 10 MB to another level. We also saw a similar pattern using an `m4.large` instance in the Amazon Ohio region. It could be some co-located batch workload lasting for hours that suddenly started or stopped. Another possible reason is that the hypervisor uses CAT to adjust the cache allocation since those instances reported CAT capable processors. We do not explore the reasons for the resource fluctuation in this work. Instead, we argue that *CacheInspector* can detect the changes and give application opportunities to adapt. Figure 9(b) shows a high discrepancy between measured L2/L3 cache sizes and those reported by the OS. We suspect that instances—specified as running on either the Intel Sandy Bridge or Broadwell platform—are actually run on servers with the new Intel Skylake platform that has 1 MB of L2 cache per core, because the results match what we measured on that platform.

We found that many L3 cache measurements in our experiment show diurnal cycles. Figure 10 shows a whole-week measurement of the L3 cache sizes using an `n1-standard-2` instance in Google Cloud. We smooth the measurement using exponential weighted moving average to filter out high-frequency components and then run fast Fourier transformation to identify the periodicity. Figure 11(a) shows that the energy concentrates around the center (low frequency). The



(a) All Frequencies (b) Frequency Zoom-in

Fig. 11. Fast Fourier transformation for an observed L3 cache size of an Google `n1-standard-2` in a week.



(a) Delta Distribution (b) Samples in one Minute

Fig. 12. Continuous L3 cache sampling with an `m4.large` instance in the Amazon Ohio Region.

zoom-in view in Figure 11(b) reveals the high energy spike at frequency 7, whose period is 1 day (one-seventh of a week). We suspect that the measured instance may be collocated with some daily workload.

We also investigated how cache allocation fluctuates in short time scales because cache allocation is less useful if it changes too fast. Therefore, we ran *CacheInspector* continuously for a day using four cloud instances as well as in a VM in our controlled environment. This experiment gave us three to five data points of L3 cache size measurements. We found that the measurements change in a range no wider than a few tens of megabytes, but they are relatively stable. Figure 12(a) shows a typical distribution of the difference between adjacent measurements. This one uses an `m4.large` instance running in the Amazon Ohio Region. The distribution fits a normal distribution centered on zero ($\sigma \approx 2.4$MB). Figure 12(b) shows a zoomed-in view, which is stable in over a minute.

Finally, we evaluated the cache size prediction error. We assume that an application invokes *CacheInspector* to sample cache allocation periodically and change its behavior accordingly. We define the prediction window as the interval of two *CacheInspector* invocations. The prediction error is the average difference between the sample and the measurements inside a window. Figure 13 shows how the prediction error grows by increasing the window size. For all tested cases, more frequent sampling results in a lower prediction error. Most of them can achieve a prediction error of 10% with a 100-second window. The Google instance (`ggl_n1s2`) displays a strong variation and hence has the highest prediction error. Since the background workload is light in the controlled environment (`server1_KVM`), its cache resource does not change frequently. Therefore, its prediction error is below 3% with hour-long windows. The prediction error is also low with Amazon m4.2xlarge instances. This instance type has four physical cores (eight virtual CPU cores) but only $\approx$13MB L3 cache as measured, which we believe is exclusive.

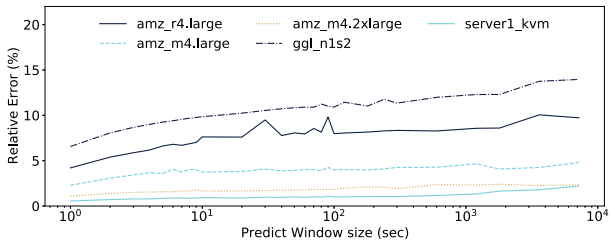


Fig. 13. L3 cache capacity prediction error.

6 RELATED WORK

Major cloud providers allow cloud users to see resource usage statistics with monitoring tools like Amazon CloudWatch, Google Stackdriver, and Azure Monitor. Although those tools provide rich statistics from hardware utilization (e.g., CPU and memory utilization per minute) to service performance (e.g., query execution time), we have not found any of them report the dynamic information of CPU cache resources, which significantly fluctuates as shown in Section 5.

Application profiling tools (a.k.a. performance profiler) report detailed information about an application during an application run [1, 31, 33, 37, 48, 58, 60]. For example, the Top-Down Characterization supported by Intel's *VTune* [37] reveals the utilization of CPU resources. Based on the event counters reported from a monitor tool called *EMON*, it pins down the performance bottlenecks like high cache miss rate, excessive TLB flushes, or too much branch misprediction. Linux *perf* is an event-based tracing infrastructure in the Linux kernel. Similar to VTune, *perf* relies on hardware performance counters. *OProfile*, *perf-tools*, and *Valgrind* are user-friendly profilers built upon *perf* [29, 41, 49]. System software developers have long used those tools to optimize algorithm and data structure for performance. However, those tools assume a static environment regarding hardware resources like CPU cache. They do not detect the changes in resource allocation, which are common in the cloud due to sharing or scheduling. Instead, *CacheInspector* detects those changes, allowing dynamic performance tuning. Moreover, *CacheInspector* does not rely on the hardware performance counters, which are rarely available from a cloud instance.

Closest to our profiling stage are CPU microbenchmarks [8, 55, 56]. STREAM has long been used for cache/memory throughput benchmarking [55, 56]. However, it assumes that the cache size is known in advance, which is invalid in the cloud. CacheBench does not assume knowing the cache size *a priori* and tests the throughput using a wide range of buffer sizes [8]. CacheBench has a feature called *guess cache size*.[3] To guess the cache size, CacheBench first measures memory throughput using a series of buffer sizes growing exponentially. Then it looks for 10% or more drops between two consecutive measurements to determine cache sizes. This approach assumes steep cliffs in Figure 2(a), which does not generally hold anymore, and we found that CacheBench is unable to determine L2/3 cache sizes. In addition, CacheBench takes minutes to find a result because it tests a wide range of buffer sizes each time—*CacheInspector* uses binary search to identify cache size in just hundreds of milliseconds. Finally, *CacheInspector* can measure cache and memory latencies; it adapts to both virtualized and bare-metal environments, which neither STREAM nor CacheBench considers.

7 CONCLUSION

We presented *CacheInspector*, a new system for measuring the cache and memory resources allocated to a user in a public cloud in a practical and online manner. Specifically, *CacheInspector*

[3]This feature is disabled by default. We enabled it for our experiments.

quantifies the capacity, throughput, and latency of each level of the memory hierarchy, allowing users to determine the best cloud offerings for their applications. Furthermore, we showed that *CacheInspector* allows users to leverage this information to adjust their application's behavior to changing, and in some cases, constrained resources in a way that avoids performance degradations. Current public clouds still offer limited visibility of architectural characteristics and allocation of shared resources to their users. *CacheInspector* is a first step toward bridging this gap between the cloud provider and cloud user's view of public cloud resources, in a way that promotes resource efficiency and performance predictability.

APPENDIX

A SELF-ADJUSTING CLOUD SERVICES

Approximate computing applications are good candidates for settings where cache allocations change over time, since they can tolerate some loss of output quality for improved execution time and/or reduced resource requirements. We use a set of machine learning Spark jobs, and Xapian, a latency-critical, search engine [43], to demonstrate their potential. We first characterize the relationship between cache capacity allocation and execution time for each examined Spark job and Xapian, as output quality decreases. We allow quality to drop by at most 5% from nominal execution (*precise*). This is achieved through several approximation techniques, namely:

- *Loop perforation*: We reduce the number of executed loops by (1) only executing the first (max_{iter}/p) iterations, and (2) executing every p^{th} iteration, or not executing every p^{th} iteration to reduce memory pressure.
- *Data type precision*: We lower the precision of suitable data types to reduce memory footprint. To identify eligible data types, we use the ACCEPT framework [66].
- *Algorithmic exploration*: We explore alternative algorithmic designs for search, sort, and graph traversal methods, which optimize for reduced resource usage instead of performance.
- *Synchronization elision*: We opportunistically execute parallel code regions without synchronization primitives like locks to avoid serialization delays and memory coherence traffic.

Once we obtain the approximate variants that are close to the performance-accuracy Pareto frontier, we modify the original application to also include calls to the approximate variants. At runtime, we use a dynamic recompilation tool, based on DynamoRIO [27], to switch between precise and approximate versions of an application, when needed. Once *CacheInspector* determines the allocated shared cache capacity, the system determines whether to lower the application's output quality or not. The prompt for a switch are Linux signals, such as SIGTERM and SIGQUIT.

Figure 14 shows an example of this operation for four Spark jobs running ML applications (SVD, PCA, k-means, and Naïve Bayes) on an Amazon EC2 m4.large instance. When *CacheInspector* determines that the application has insufficient cache resources to meet its performance requirements, DynamoRIO invokes the least aggressive approximate variant of the application that allows it to meet its QoS. When cache resources return to nominal levels, the job switches back to precise execution. For all four jobs, inaccuracy is 2.1% on average and never exceeds the 5% threshold. The vertical dotted lines show the required completion time for each job, whereas the solid line of the same color shows when the job would complete, if approximation was not employed. The bottom of Figure 14 also shows the process for an interactive, latency-critical service. Inaccuracy again is adjusted based on the available cache resources, ensuring that the tail latency QoS is met at all times.

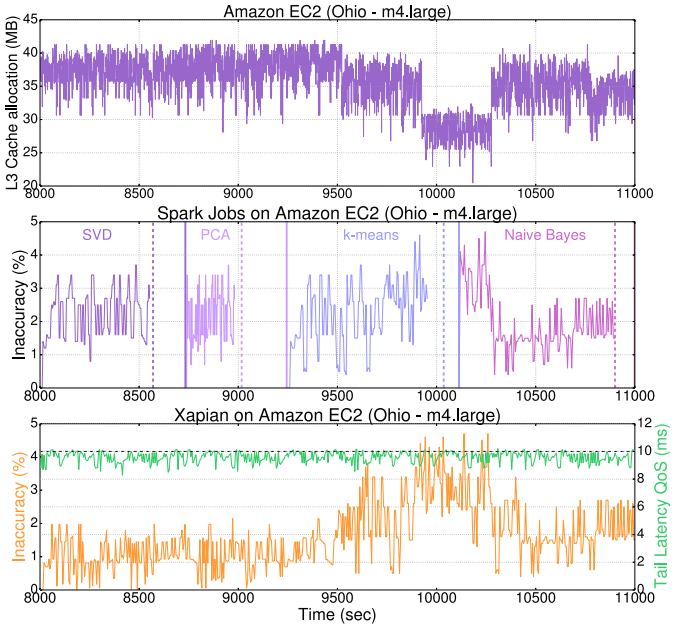


Fig. 14. Switch between precise and approximate execution for several Spark jobs (top) and the Xapian [45] search engine (bottom) under varying L3 cache allocations. When the allocated cache is below the requirement for nominal performance, the system sacrifices some output quality for reduced resources and improved execution time.

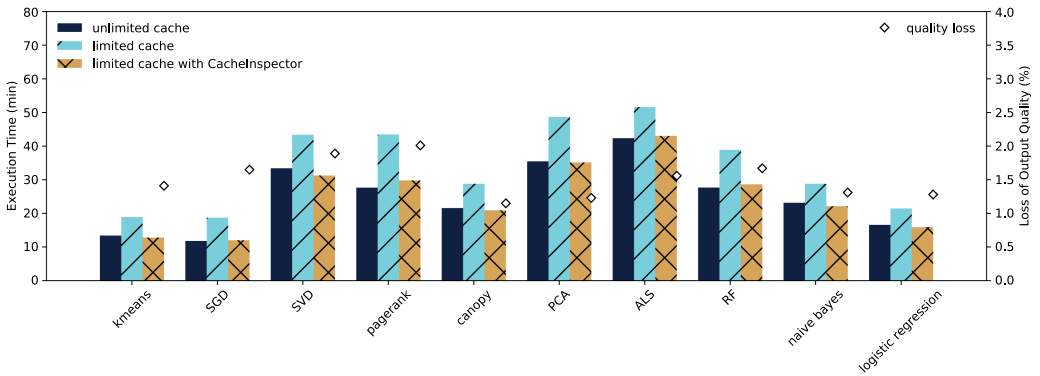


Fig. 15. Impact of reverse engineering the effective cache resources in a public cloud with *CacheInspector*.

In general, dynamic recompilation can introduce significant runtime overheads if invoked at the granularity of individual instructions. To avoid such overheads, DynamoRIO is only invoked at coarse, function granularity, which allows it to always incur less than 5.6% overhead in execution time.

Figure 15 shows the impact of *CacheInspector*'s measurements on a wider range of application. In this experiment, we run a set of analytics applications from Spark's MLlib framework in a two-socket server, shared among multiple applications, and equipped with Intel's CAT mechanism. The leftmost bars show each job's performance when cache resources are unlimited, whereas the light blue bars show the execution time per job when cache resources are limited, due to con-

tention. All applications experience significant performance degradation—27% on average and up to 45%. Unfortunately, in this case, there is little the user can do to reverse the impact of resource contention, beyond requesting more resources, and/or migrating to a different server platform, or cloud provider altogether. In contrast, when *CacheInspector* informs the user of the instantaneously available cache resources, the application can be adjusted to avoid the impact of the degraded performance. In all cases, the Spark applications tolerate 1% to 2% of loss in their output quality[4] in return for completing at the same time as when cache resources were plentiful.

ACKNOWLEDGEMENTS

We thank Ken Birman, Yu Tao, and anonymous reviewers for their insightful and helpful feedback.

REFERENCES

[1] AMD. n.d. AMD uProf. Retrieved April 3, 2021 from https://developer.amd.com/amd-uprof/.
[2] Weijia Song. n.d. CacheInspector GitHub repository. Retrieved April 3, 2021 from https://github.com/songweijia/CloudModeling.
[3] Die.net. n.d. clock_gettime, Linux man page. Retrieved February 15, 2017 from https://linux.die.net/man/3/clock_gettime.
[4] Intel. n.d. Higher Performance When You Need It Most. Retrieved February 28, 2017 from http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.
[5] AWS. n.d. How Do I Configure HugePages on My Amazon EC2 Linux Instance? Retrieved April 3, 2021 fromhttps://aws.amazon.com/premiumsupport/knowledge-center/configure-hugepages-ec2-linux-instance/.
[6] 7-cpu.com. n.d. Intel Haswell. Retrieved February 28, 2017 from http://www.7-cpu.com/cpu/Haswell.html.
[7] 7-cpu.com. n.d. Intel Sandy Bridge. Retrieved December 5, 2017 from http://www.7-cpu.com/cpu/SandyBridge.html.
[8] UTK. n.d. LLCBench. Retrieved May 13, 2017 from http://icl.cs.utk.edu/llcbench/index.htm.
[9] Microsoft. n.d. Set Up DPDK in a Linux Virtual Machine. Retrieved April 3, 2021 from https://docs.microsoft.com/en-us/azure/virtual-network/setup-dpdk#set-up-the-virtual-machine-environment-once.
[10] Intel. 2017. Intel® 64 and IA-32 architectures software developer's manual. Volume 1: Basic Architecture.
[11] Intel. 2017. Intel® 64 and IA-32 Architectures Optimization.
[12] Intel. 2017. Intel® 64 and IA-32 architectures software developer's manual. Volume 3: System Programming Guide.
[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. 265–283.
[14] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting hardware-assisted page walks for virtualized systems. *ACM SIGARCH Computer Architecture News* 40 (2012), 476–487.
[15] Luiz Barroso and Urs Hoelzle. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. MC Publishers.
[16] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*.
[17] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA'15)*.
[18] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. *ACM SIGPLAN Notices* 43, 3 (March 2008), 26–35. DOI: https://doi.org/10.1145/1353536.1346286
[19] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious structure layout. *ACM SIGPLAN Notices* 34, 5 (May 1999), 1–12. DOI: https://doi.org/10.1145/301631.301633
[20] Stephanie Coleman and Kathryn S. McKinley. 1995. Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices* 30, 6 (June 1995), 279–290. DOI: https://doi.org/10.1145/223428.207162
[21] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 308–319. DOI: https://doi.org/10.1145/2485922.2485949

[4]The exact function of output quality differs across applications. For example, for *k-means*, it is the $|L_2|$ norm for the distance of elements in a cluster.

[22] David Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing.

[23] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*.

[24] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[25] Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. Red Hat Inc.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, et al. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19.

[27] DynamoRIO. n.d. DynamoRIO: Dynamic Instrumentation Tool Platform. Retrieved April 3, 2021 from http://www.dynamorio.org.

[28] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*. 226–231. http://dl.acm.org/citation.cfm?id=3001460.3001507

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

[30] Jeff Gilchrist. n.d. Parallel BZIP2 (PBZIP2) Data Compression Software. Retrieved October 11, 2017 from http://compression.ca/pbzip2/.

[31] Francis Giraldeau, Julien Desfossez, David Goulet, Mathieu Desnoyers, and Michel R. Dagenais. 2011. Recovering system metrics from kernel trace. In *Proceedings of the 2011 Linux Symposium*.

[32] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, New York, NY, 22.

[33] Brendan Gregg. n.d. Linux Performance. Retrieved July 31, 2018 from http://www.brendangregg.com/linuxperf.html.

[34] Brendan Gregg. 2017. The PMCs of EC2: Measuring IPC. Retrieved April 3, 2021 from http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html.

[35] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, et al. 2015. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 223–238. DOI: https://doi.org/10.1145/2694344.2694347

[36] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, Los Alamitos, CA, 657–668.

[37] Intel. n.d. Getting Started with Intel VTune Amplifier 2018. Retrieved July 31, 2018 from https://software.intel.com/en-us/get-started-with-vtune.

[38] Intel. 2011. *Intel Math Kernel Library Reference Manual*. Technical Report 630813-051US. Available at http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf.

[39] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 151–162.

[40] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Computer Architecture News* 38, 3 (June 2010), 60–71. DOI: https://doi.org/10.1145/1816038.1815971

[41] Xinxin Jin, Haogang Chen, Xiaolin Wang, Zhenlin Wang, Xiang Wen, Yingwei Luo, and Xiaoming Li. 2009. A simple cache partitioning approach in a virtualized environment. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, Los Alamitos, CA, 519–524.

[42] Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 729–742.

[43] Harshad Kasture and Daniel Sanchez. 2016. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*.

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP'19)*. IEEE, Los Alamitos, CA, 1–19.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. 1097–1105.

[46] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. *ACM SIGARCH Computer Architecture News* 19 (1991), 63–74.

[47] Michael A. Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. 2014. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. IEEE, Los Alamitos, CA, 558–570. DOI: https://doi.org/10.1109/MICRO.2014.21

[48] John Levon. n.d. OProfile. Retrieved July 31, 2018 from http://oprofile.sourceforge.net/credits/.

[49] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. 1997. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*. IEEE, Los Alamitos, CA, 213–224.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arXiv:1801.01207.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*.

[52] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[53] Chris Lomont. 2011. *Introduction to Intel Advanced Vector Extensions*. White Paper 23. Intel.

[54] Jason Mars and Lingjia Tang. 2013. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.

[55] John D. McCalpin. 1991–2007. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Technical Report. University of Virginia, Charlottesville, VA. (A continuously updated technical report: http://www.cs.virginia.edu/stream/).

[56] John D. McCalpin. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[57] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. 237–250.

[58] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices* 42 (2007), 89–100.

[59] Venkatesh Pallipadi. 2009. *Enhanced Intel Speed Step Technology and Demand-Based Switching on Linux*. Intel Developer Service.

[60] Vara Prasad, William Cohen, F. C. Eigler, Martin Hunt, Jim Keniston, and J. Chen. 2005. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium*. 49–64.

[61] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, and Calton Pu. 2010. Understanding performance interference of I/O workload in virtualized cloud environments. In *Proceedings of the IEEE 3rd International Conference on Cloud Computing (CLOUD'10)*. IEEE, Los Alamitos, CA, 51–58.

[62] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. 2013. Who is your neighbor: Net I/O performance interference in virtualized clouds. *IEEE Transactions on Services Computing* 6, 3 (2013), 314–329.

[63] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM, New York, NY, 381–391. DOI: https://doi.org/10.1145/1250662.1250709

[64] Venu Gopal Reddy. 2008. *Neon Technology Introduction*. ARM Corporation.

[65] Charles Reiss, Alexey Tumanov, Gregory Ganger, Randy Katz, and Michael Kozych. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. 1–13.

[66] Adrian Sampson, Andre Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. n.d. *ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing*. Technical Report UW-CSE-15-01-01. University of Washington.

[67] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*.

[68] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. *ACM SIGARCH Computer Architecture News* 39 (2011), 57–68.

[69] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. 351–364.

[70] Timothy Sherwood, Brad Calder, and Joel Emer. 1999. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*. ACM, New York, NY, 155–164.

[71] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W'11)*. IEEE, Los Alamitos, CA, 194–199.

[72] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Jenga: Software-defined cache hierarchies. In *Proceedings of the 44th International Symposium in Computer Architecture (ISCA'17)*.

[73] ARM Developer. n.d. ARM Architecture. Retrieved May 16, 2018 from http://infocenter.arm.com/help/index.jsp.

[74] Kernel. n.d. hugetlbpage.txt. Retrieved April 3, 2021 from https://bit.ly/2LCLaw8.

[75] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely Jr., and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 430–441. DOI: https://doi.org/10.1145/2155620.2155671

[76] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. 2017. vCAT: Dynamic cache management using CAT virtualization. In *Proceedings of the 2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'17)* IEEE, Los Alamitos, CA, 211–222.

[77] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. 719–732.

Received July 2020; revised January 2021; accepted March 2021