Measuring Data Access Latency in Large CPU Caches

Shaotong Sun
University of Rochester
United States
ssun25@u.rochester.edu

Yifan Zhu
University of Rochester
United States
yifanzhu@rochester.edu

Xingzhi Ye
University of Rochester
United States
xye8@u.rochester.edu

Chen Ding
University of Rochester
United States
cding@cs.rochester.edu

Abstract

This paper describes a new, multi-locality benchmark program for testing memory access latency and using it to study recent AMD machines equipped with 3D vertical cache (V-Cache) that can be over 1 GiB in total size on a single node. The latency study shows that these large caches differ from traditional LLCs in two aspects: the V-Cache is partitioned rather than shared, and the cache replacement policy is more similar to random than it is to LRU.

CCS Concepts

• **Computer systems organization** → *Multicore architectures*; • **General and reference** → **Empirical studies**; **Measurement**; **Evaluation**; **Experimentation**; *Performance*.

Keywords

Cache, CPU, Profiling, V-Cache, Replacement Policy

ACM Reference Format:
Shaotong Sun, Yifan Zhu, Xingzhi Ye, and Chen Ding. 2024. Measuring Data Access Latency in Large CPU Caches. In *The International Symposium on Memory Systems (MEMSYS '24), September 30–October 03, 2024, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3695794.3695806

1 Introduction

In a modern computing system, the disparity of speed in cores and memory significantly impacts program performance in both speed and energy consumption, known as the memory wall [10] [18]. With the number of cores growing, the data consumption of the machine becomes larger and faster, and data movement is becoming more and more important in program performance. Therefore, programs and algorithms must be carefully designed to minimize data movement, which raises the importance of data locality analysis.

Data locality cost can be expressed in many different ways. One of the easiest way to measure data movement cost is through access latency. In other word, by measuring the average data access latency, we can quantify the cost of data movement and the benefit of caching.

Our contributions are

- specifications of recent AMD machines with up to 768 MiB V-Cache on a single machine
- a benchmark program that tests access latency on data traversals that have different data reuse patterns:
  (1) Cyclic: data is repeatedly traversed in the same order
  (2) Sawtooth: the order is reversed each time the data is traversed
- results from V-Cache based machines and a comparison with other systems

The results show that V-Cache differs from conventional LLCs in two ways:

(1) The replacement policy is clearly different from the one used in the L2 cache on the same machine and the policy used on an Intel machine.
(2) Only local V-Cache is used by a core, and having more V-Cache modules does not help if only a single core is used, i.e., partitioned LLC rather than shared LLC.

The replacement policy is a major factor in determining the average access latency, but it is not disclosed by AMD. The purpose of using two types of traversals is to (1) identify how LRU like the policy is and (2) how the policy is sensitive and insensitive to the order of data traversal.

A limitation is that the new test works on power of two array sizes. The effect of caching is measured by the cumulative time of data traversals, not the number of cache misses, which require access to cache hardware, e.g., hardware counters. As a result, the test is portable and does not depend on special hardware support or correctly understanding such support.

2 Dual Access Order Stream-in-Cache Benchmark (SiC)

On modern computers, the access latency is made obscure by non-data access costs including instructions and auxiliary data, latency hiding through out-of-order execution and prefetching, contention on data bandwidth, and cache coherence. When the data size is greater than the cache size, the latency depends on the miss ratio. This section presents the pseudo code and the techniques for designing such a test program that overcomes these difficulties and accurately measures the latency of data access.

MEMSYS '24, September 30–October 03, 2024, Washington, DC, USA

ACM ISBN 979-8-4007-1091-9/24/09
https://doi.org/10.1145/3695794.3695806

2.1 Triangular Access

One of the major problem for creating an access latency benchmark is prefetching. Prefetcher will automatically fetch the data forward, thus making the timing ineffective. One common method people use when benchmarking latency is to manually disable hardware prefetcher, in BIOS or in Machine Specific Registers (MSR). However, in this work, by utilizing the power of Triangular Sequence, it is no longer needed for manually disabling hardware prefetcher.

Triangular Sequence is a common practice among the hash table community. This is a standard sequence used in quadratic probing of open hash tables [9]. It is well-known that the triangular sequence can access each number from 0 to $2^m - 1$ exactly once. In other word, $f(x) = \sum_{n=1}^{x} n \mod 2^m$ generates a permutation on $\{0, ..., 2^m - 1\}$ [2][1]. This unique feature allows one to easily generate a pseudo random traversal for some given arrays, and it is complex enough to defeat the hardware prefetcher, but, at the same time, computationally light enough to not be a bottleneck for our benchmark. However, one limitation of using the triangular access is that we can only bench data size that is 2^n.

One thing to be noted is that by enforcing the triangular sequence access, this benchmark mimics the probing process of a large hash table in CPUs, such as Google's sparse hash table[1] and Swiss table[7], thus characterizes the performance of such tables with large on-chip caches.

2.2 Dependent Loads

Modern processors employ latency hiding techniques such as prefetching and non-blocking loads. To remove such effects, we convert all data loading into dependent loads in a loop. In this way, the next load cannot start until the previous load has already completed. It is to be noted that although data dependence requires a write, in this benchmark, the write would be a write towards a register instead of an array address. The memory access consists entirely of dependent loads.

2.3 Sawtooth and Cyclic Data Traversals

The SiC benchmark uses repeated data traversals in two different orders. In *Cyclic*, the order of data traversal stays the same. In *Sawtooth*, the order is reversed at each traversal. Sawtooth performs the best in caches using Least Recently Used (LRU) replacement policy, and Cyclic performs worst in LRU, while both of the traversals perform around the same under Most Recently Used (MRU) replacement policy. With this enabled, one can see the performance results and determine the cache replacement policy based on the results. Specifically, in the results, `for_for` and `back_back` curves are Cyclic traversal, and `for_back` is the Sawtooth traversal. To measure the latency of accessing different levels of cache, we need to account for the cache replacement policy. While it is non-trivial to determine the exact replacement policy, we measure how a cache performs relatively for Sawtooth and Cyclic.

Triangular access is not a new technique. However, using it with dependent loads is a novel extension, so is the ability to traverse in opposite orders without adding new memory access. Algorithms 1 shows the pseudo code for the benchmark, and Algorithms 3 and 4 show the code for `for_for` cyclic traversal and `for_back` sawtooth traversal respectively. For dependent loads, they need to pre-fill the arrays with the index of the next access, which is shown in Algorithm 2. The code for `back_back` is similar.

[1]Prove can be find at: https://fgiesen.wordpress.com/2015/02/22/triangular-numbers-mod-2n/ or *The Art of Compute Programming*, Volume 3, Chapter 6.4

Algorithm 1 Benchmark

```
1: procedure BENCH(input : array size)
2:     set thread affinity to CPU0                    ▷ core binding
3:     allocate memory block          ▷ huge page (if applicable)
4:     PREFILL (Algo. 2) test array with triangular numbers
5:     clear cache
6:     load the memory block into the cache
7:     load data with different PATTERNS (Algo. 3, 4)
8: end procedure
```

Algorithm 2 Prefill Array

```
1: procedure PREFILL(input : array size)
2:     for each elements i in a do
3:         a[i] ← array size + 1
4:     ▷ all array elements are 8 bytes, meaning every cache line
   contains only 8 elements
5:     end for
6:     p ← 0
7:     stride ← 0
8:     tri ← 0
9:     for p < array size do                          ▷ p += 8
10:        t_k ← stride + tri
11:        tri ← t_k  mod array size                  ▷ triangular access
12:        a[tri] ← tri
13:        stride ← stride + 8
14:    end for
15: end procedure
```

2.4 Core Binding

In almost all operating systems, the scheduler would sometimes swap processes around different cores to balance the work. This happens especially frequently in desktop and real-time devices, and might also happen in a server environment if the program triggers a preemption point [12]. To ensure that SiC test the same caches through its execution, we bind the benchmark to a physical core.

2.5 Cache Line Access

In modern processors, the entire cache line (64 bytes) is fetched when we access any part of that cache line, which means that if contiguously access the rest of the cache line, the latency result will be mitigated by latency of the already fetched cache line. In order to accurately calculate the latency per touch (fetch), we decided to access exactly one time per cache line. In this way, despite the processor will fetch all 64 bytes (the whole cache line), we would still get a accurate result of the latency data.

2.6 Huge Pages

In Zen3 and Zen4 Microarchitectures, the Data Translation Lookaside Buffer (DTLB) sizes are 64 entries (L1), 2048 entries (L2) and

Algorithm 3 Forward Forward Pattern (Cyclic traversal)

```
1:  procedure FORFOR(input : array size)
2:      p ← 0
3:      stride ← 0
4:      tri ← 0
5:      timer start here
6:      for p < array size do                          ▷ p += 8
7:          t_k ← stride + tri
8:          tri ← t_k mod array size                   ▷ triangular access
9:          tri ← a[tri]                               ▷ dependent load
10:         stride ← stride + 8                        ▷ per cache line access
11:     end for
12:     p ← 0
13:     stride ← 0
14:     tri ← 0
15:     for p < array size do                          ▷ p += 8
16:         t_k ← stride + tri
17:         tri ← t_k mod array size                   ▷ triangular access
18:         tri ← a[tri]                               ▷ dependent load
19:         stride ← stride + 8                        ▷ per cache line access
20:     end for
21:     timer end here
22: end procedure
```

Algorithm 4 Forward Backward Pattern (Sawtooth traversal)

```
1:  procedure FORBACK(input : array size)
2:      p ← 0
3:      stride ← 0
4:      tri ← 0
5:      timer start here
6:      for p < array size do                          ▷ p += 8
7:          t_k ← stride + tri
8:          tri ← t_k mod array size                   ▷ triangular access
9:          tri ← a[tri]                               ▷ dependent load
10:         stride ← stride + 8                        ▷ per cache line access
11:     end for
12:     p ← 0
13:     for p < array size do                          ▷ p += 8
14:         t_k ← tri − stride + array size
15:         tri ← t_k mod array size                   ▷ triangular access
16:         tri ← a[tri]                               ▷ dependent load
17:         stride ← stride − 8                        ▷ per cache line access
18:     end for
19:     timer end here
20: end procedure
```

72 entries (L1), 3072 entries (L2), respectively. This means that, sometimes, in order to test the large LLC, we will exceed the TLB capacity if we use the default 4 KiB pages. To solve this issue, we have employed huge pages up to 1 GiB instead of the standard 4 KiB pages.

3 Evaluation

This section shows the test results using the SiC benchmark and the analysis of the AMD L3 caches based on these results, including the usable L3 cache size per core and the replacement policy.

3.1 Machine Specifications

We have tested two AMD machines: one is server class (EPYC), and the other is a desktop (Ryzen). For comparison, we have also tested an Intel server machine (Xeon). As listed on their product pages, the L3 cache is 768 MiB and 128 MiB for the two AMD machines and 15 MiB for the Intel machine.

The first machine is AMD EPYC 7773X, launched in March 2021. The specification is as follows based on the company product page[2]:

- Up to 3.5 GHz
- Zen 3 Microarchitectures
- 8 Core Chiplet Dies (CCDs), each contains:
  - 8 cores 16 threads
  - L1i: 256 KiB(8 × 32 KiB) L1d: 256 KiB(8 × 32 KiB) 8-way set associative (ECC) (write-back)
  - L2: 0.5 MiB (8 × 512 KiB) 8-way set associative (ECC) (write-back)
  - L3: 768 MiB total size, 32 MiB + 64 MiB 3D V-Cache per CCD 16-way set associative (ECC) (L2 Victim Cache) (write-back)

The second machine is AMD R9 7950X3D, launched in February 2023[3]:

- Up to 5.7 GHz
- Zen4 Microarchitectures
- 2 CCDs, each contains:
  - 8 cores 16 threads
  - L1i: 256 KiB(8 × 32 KiB) L1d: 256 KiB(8 × 32 KiB) 8-way set associative (ECC) (write-back)
  - L2: 1 MiB (8×1024 KiB) 8-way set associative (ECC) (write-back)
  - L3: 128 MiB total size, 32 MiB + 64 MiB 3D V-Cache only on CCD0 16-way set associative (ECC) (L2 Victim Cache) (write-back)

The third machine is Intel(R) Core(TM) i7-6700, lauched in 2015[4]:

- Up to 4.0 GHz
- Skylake Microarchitectures
  - 4 cores 8 threads
  - L1i: 128 KiB (4 × 32 KiB) L1d: 128 KiB (4 × 32 KiB)
  - L2: 1 MiB (4 × 256 KiB)
  - L3: 8 MiB

The forth machine is Intel(R) Xeon(R) Gold 6230, lauched in 2019[5]:

- Up to 3.9 GHz
- Cascade Lake Microarchitectures
- 2 NUMA nodes, each contains:
  - 20 cores 40 threads

[2] https://www.amd.com/en/product/11851
[3] https://www.amd.com/en/product/12741
[4] https://ark.intel.com/content/www/us/en/ark/products/88196/intel-core-i7-6700-processor-8m-cache-up-to-4-00-ghz.html
[5] https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

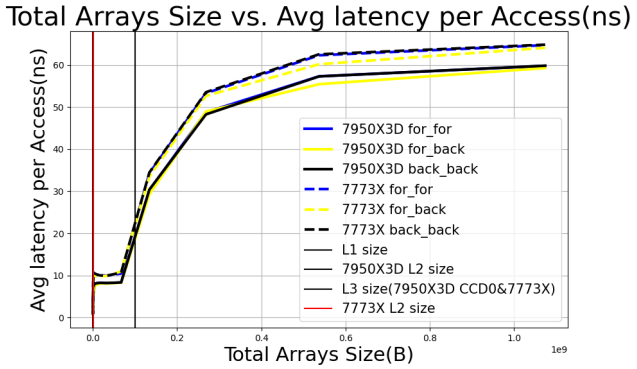

Figure 1: Access latency in nanoseconds comparison for EPYC 7773X and R9 7950X3D

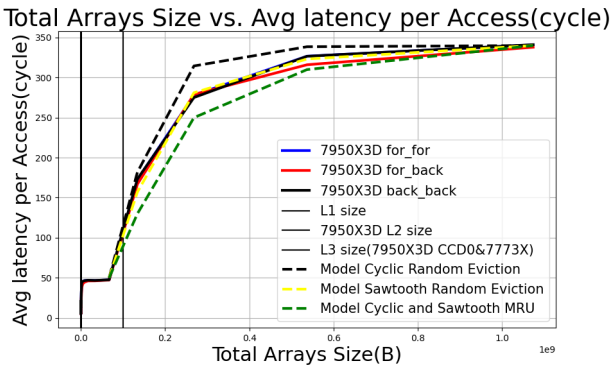

Figure 2: Access latency in cycles on L3 R9 7950X3D, compared with the calculated miss ratio of MRU and random cache replacement policy

  - L1i: 640 KiB (20 × 32 KiB) 8-way set associative L1d:640 KiB (20 × 32 KiB) 8-way set associative write-back
  - L2: 20 MiB (20 × 1 MiB) 16-way set associative write-back
  - L3: 27.5 MiB (20 × 1.375 MiB) 11-way set associative write-back

The fifth machine is Intel Xeon E5-2430, introduced in 2011. The machine has two NUMA nodes. Each node has the following specification[6]:

- Up to 2.7 GHz
- Sandy Bridge-EN Microarchitectures
  - 6 cores 12 threads
  - L1i: 192 KiB (6 × 32 KiB) L1d: 192 KiB (6 × 32 KiB)
  - L2: 1.5 MiB (6 × 256 KiB)
  - L3: 15 MiB

3.2 V-Cache Partitioning

We have experimented with data size up to 1 GiB, large enough to include the full effect of the L3 V-Cache. It shows the range of the access latency from when the data fits in the cache, i.e., the miss ratio is zero, to the data size when most accesses would be cache misses.

Figure 1 shows the access latency for the AMD desktop machine (same as Figure 2) in nanoseconds. To convert from cycle count to nanoseconds, we use the peak core frequency for each core since this benchmark is bound to one core, assuming that the CPU runs at the peak frequency to boost performance. The actual frequency may vary during the execution. The y-axis shows the access latency in nanoseconds. One can also see that although the latency results in these two processors vary, though the shape of the curves are identical.

The AMD server machine has 768 MiB L3 in total, while the desktop machine has 128 MiB. We would expect that when the data size exceeds the smaller L3 (128 MiB), it still fits in the larger L3 (768 MiB). The larger L3 should not have the dramatic latency increase that the smaller L3 had when accessing same amount of data. In other words, if the whole L3 is shared, accessing data size that is bigger than the smaller L3 but smaller than the larger L3 (such as 256 MiB) by the 7773X (with the larger L3) should not incur a dramatic latency increase as if accessing the same data through 7950X3D (with the smaller L3).

However, on both machines, the most abrupt latency increase happens between the same data sizes, from 64 MiB to 128 MiB. On both machines, the local V-Cache on each CCD is 96 MiB (in the case of non-uniform cache placement, we bench on the CCD that have a larger cache; i.e. we benchmarked on CCD0 in 7950X3D). On each machine, three different traversal orders show the same effect. In all six cases, the latency result shows that the data size fits in the cache at 64 MiB but not at 128 MiB. Therefore, we conclude that *the L3 cache on AMD CCDs are partitioned.* Each CCD has 96 MiB V-Cache. A program run on one CCD cannot make use of the L3 on other CCDs.

From the perspective of L3, CCDs on AMD machines are similar to nodes on conventional multi-socket machines. Note that CCDs are not traditional nodes in that CCDs share the same memory interface while traditional nodes have their own memory interfaces. Nodes have non-uniform memory access (NUMA).

Since both machines show similar access latency, we next study the desktop machine and analyze the V-Cache cache replacement policy.

3.3 V-Cache Is Not LRU

On V-Cache, we found that Cyclic and Sawtooth have similar latencies. This is shown in Figure 2, where the latency curves of `for_for`, `for_back`, and `back_back` largely overlap. This indicates that V-Cache does not use LRU policy. In this section, we consider the possibility of MRU and random replacement policy being used by V-Cache.

3.3.1 Model and Simulation. In Figure 2, we show a modeled Sawtooth random eviction miss ratio curve and a modeled Cyclic random eviction miss ratio curve alongside with the access latency curve of the R9 7950X3D. Random eviction in cache can be easily modeled and has been used in StatCache [8] and the analysis of cache sharing [17]. Using the binomial model, one can show that

[6]https://www.intel.com/content/www/us/en/products/sku/64616/intel-xeon-processor-e52430-15m-cache-2-20-ghz-7-20-gts-intel-qpi/specifications.html

for the Cyclic order, the random eviction miss ratio curve is as follows. Let m be the data size in the number of cache blocks, and c the cache size in the number of cache blocks. We have:

$$mr(m,c) = 1 - (1 - \frac{1}{c})^{m*mr(m,c)}$$

The formula differs from the usual miss ratio function in that it has both the cache size and the data size. It is also recursive. We solve it for specific m, c values numerically.

Using the same binomial model, the random eviction miss ratio for Sawtooth is

$$mr(m,c) = 1 - (1 - \frac{1}{c})^{\frac{1}{m}\sum_{i=1}^{m}(2i-1)*mr(m,c)}$$

which comes to the exact same miss ratio as Cyclic. The mathematics shows that Cyclic and Sawtooth have the same miss ratio in a single level random eviction cache.

The prior work by Ye et al. [19] studied AMD and Intel machines and modeled LRU for the L3 cache on both. Here we extend the model to account for random replacement in L3 and compare it with measurements on the new AMD V-Cache.

Intuitively, every data elements in Cyclic order having equal chance being in the cache later in the traversal, while, in Sawtooth order, the data closer to the turning point (the point where the reverse access begin) of the traversal have a higher chance being a cache hit than those data that's further away from the turning point. For example, for cache size equals to 2, in a Cyclic trace *abcabc*, the possibility of the second *a*, *b*, or *c* being in the cache when needed is the same; in a Sawtooth trace *abccba*, however, the possibility of the second *c* is a cache hit is 100%, and the possibility decreases as the data element is further from the turning point.

To better model the L3 cache of the R9 7950X3D, which uses a three-level cache and an exclusive cache hierarchy for L3, assuming L1 and L2 use LRU like cache replacement policy and L3 uses random like replacement policy, we adjust the above formulas. For Cyclic, each data block, after it is accessed, first goes to L1, then evicted from L1 to L2, and finally from L2 to L3. During the time it is in L1 and L2, it is not at risk of being evicted from L3. We call it the *shielding* effect. The reason is that during a reuse interval, in a part of the period, the object is in L1 or L2, so it is shielded from random eviction in L3. However, we believe that *shielding* effect would not affect the miss ratio of the L3 cache:

$$mr(m,c_2,c_3) = 1 - (1 - \frac{1}{c_3})^{m*mr(m,c_2,c_3)}$$

For Sawtooth, there is cache reuse at the boundary between two repeats, and the number of L1 and L2 cache hits is c_2 for each repeat. These hits are subtracted to obtain the actual reuse interval for L3. In the non-boundary parts, the accesses are cache misses. Unlike the Cyclic pattern, the Sawtooth pattern has uneven reuse intervals. When an exclusive L3 cache is introduced, separating L1 and L2 caches, the blocked reuse intervals at the boundary are the larger reuse intervals. In other words, the cache reuse at the boundary is exactly as mentioned above, except the number of reuses is elongated to the size of c_2

The combined adjustment produces the following:

$$mr(m,c_2,c_3) = 1 - \frac{1}{m-c_2}\sum_{i\in RI_s}(1 - \frac{1}{c_3})^{i*mr(m,c_2,c_3)}$$

We adjusted to cache block granularity in order to better model the actual result. All above are also taken into account in the random eviction simulator.

3.3.2 Actual Results from SIC Benchmark. Figure 2 shows the miss ratio at cache size of 96 MiB and how it varies when the data size increases to 1 GiB. Note that the miss ratio curve does not readily convert to latency. They have different units. The y-axis shows the access latency. The miss ratio, not shown in the y-axis, increases from 0% to 100%.

We see that the replacement policy is unlikely LRU. The reason is that Cyclic accesses, which include the `for_for` and `back_back` in the figure, have near identical latency as Sawtooth access, which is the `for_back` order. If the replacement policy is LRU, we would see large differences between these two types of accesses, since Sawtooth accesses have better locality in the LRU cache than Cyclic accesses have. More detailed data points will be given in later paragraphs.

The shape of the miss ratio curve is compared with that of the latency curve. While the match is not very close for the modeled Cyclic curve, the actual result is nearly identical to the modeled Sawtooth curve, as shown in Figure 2. This is one data point that suggests that the replacement policy may be more similar to random.

3.3.3 MRU Model and Comparison. In Figure 2, we also demonstrate a modeled MRU curve alongside with all other curves. Let m be the data size in the number of cache blocks, and c the cache size in the number of cache blocks. One can deduce that for Cyclic and Sawtooth access patterns, the expected miss ratio is

$$mr(m,c) = 1 - \frac{c}{m}$$

As shown in 2, the actual latency curves for Cyclic and Sawtooth patterns are similar, which indicates that MRU could be a viable candidate for the unknown replacement policy on the AMD CPU. However, the modeled Cyclic and Sawtooth MRU curve has a considerable gap between its curve and the actual benchmarked curves, suggesting that the cache replacement policy might not be MRU-like.

3.3.4 Random Eviction Model Implementation. Both Cyclic and Sawtooth models use recursive equations. The miss ratio is the fixed-point solution of these equations. We have implemented the model in Rust[7] and Python by solving the recursive equations using the Newton method.

To verify its correctness, we have also implemented a simulator for random replacement cache in Rust.[8] We found that for the cache size we modeled, e.g., 1 GiB cache which has 16 million cache blocks, the model result could be 40% lower than the simulation result for Cyclic but 50% higher for Sawtooth. These errors disappear once we switch from using single-precision floating-point (`f32` in Rust) to double-precision floating-point (`f64`).

Next we test Cyclic and Sawtooth traversal orders in other caches on AMD and caches on Intel processors.

[7]The source code of the Rust solution is available at https://github.com/sauceeeeage/newton4corun.

[8]The Rust source code of the simulator is available at https://github.com/XingzhiY/random_eviction_cache.

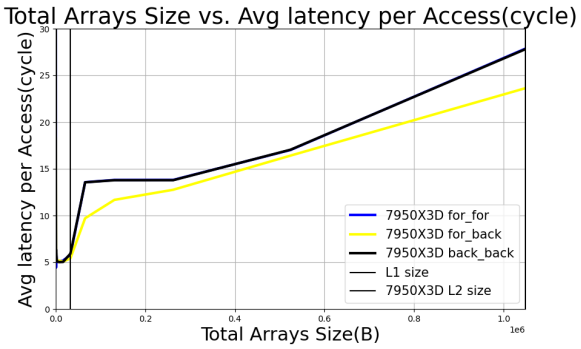

Figure 3: R9 7950X3D at L1

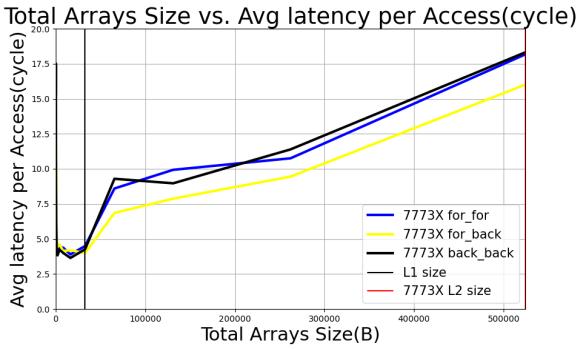

Figure 4: EPYC 7773X at L1

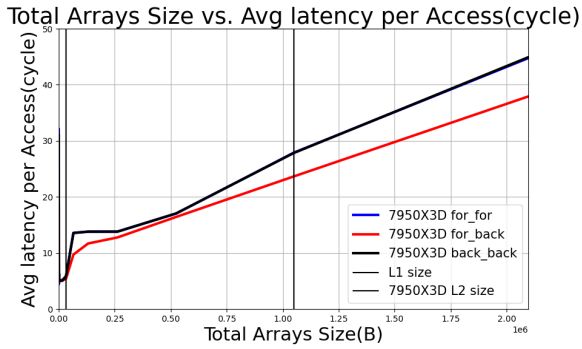

Figure 5: R9 7950X3D at L2

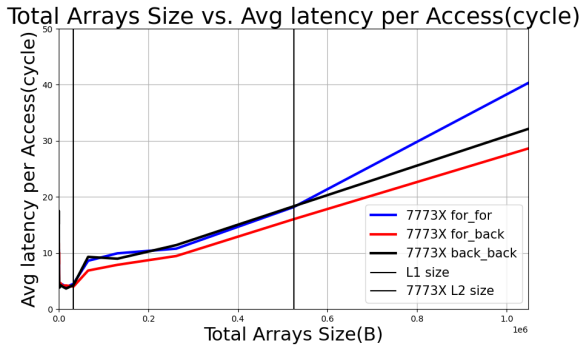

Figure 6: EPYC 7773X at L2

3.4 AMD Cache Analysis

3.4.1 L1 and L2. In a LRU cache, the Sawtooth order of traversal ought to have better locality than that of the Cyclic. In Figure 3 and Figure 4, we tested both AMD machines for 64 KiB, which is the closest data size that is larger than the L1 cache size (32 KiB). In Figure 5 and Figure 6, similarly, we tested to the closest data size that is larger than the size of the L2 cache, which is 1 MiB on R9 7950X3D and 512 KiB on EPYC 7773X.

Table 1 shows all of the data points of AMD CPUs plotted in graphs.

For example, Figure 3 shows the latency of three traversal orders on R9 7950X3D at L1. For the Sawtooth traversal, the latency for array size 64 KiB is 9.7 cycles. For the two Cyclic traversal orders, the latencies are both 13.7 cycles. The latency of Sawtooth is lower than that of cyclic by

$$\frac{|\text{Sawtooth latency} - \text{Cyclic latency}|}{\text{Cyclic latency}}$$

which is

$$\frac{|9.7 - 13.7|}{13.7} \approx 0.29$$

Similarly, we apply the above formula to all data points.

On both machines, Sawtooth is clearly faster than Cyclic, which is expected if the cache replacement policy is LRU.

3.4.2 L3 V-Cache. In contrast, the two access orders show similar access latency in L3 V-Cache. In Figure 2, for Sawtooth, the latency at array size 128 MiB (the smallest size that is greater than L3 size) is 166.8 cycles, and for cyclic, the latency is 172.8 cycles. The latency of Sawtooth is lower than that of Cyclic by

$$\frac{|166.8 - 172.8|}{172.8} \approx 0.03$$

Therefore, as the performance result suggested, the cache replacement policy for L1 and L2 cache in 7950X3D and 7773X should be LRU. On the contrary, the L3 cache replacement policy in both 7950X3D and 7773X should not be LRU.

3.5 Intel Cache Analysis

In this section, we analyze the cache replacement policies of three Intel CPUs: i7-6700(in Figure 7, 10, and 15), Xeon Gold 6230(in Figure 8, 11, and 14), and Xeon E5-2430(in Figure 9,12 , and 13). We apply the same methodology as in section 3.4 on AMD CPUs, examining the latencies for Sawtooth and Cyclic traversal orders. Table 2 shows all of the data points of Intel CPUs plotted in graphs.

3.5.1 L1, L2, and L3. As shown in Figure 14, unlike R9 7950X3D and EPYC 7773X, both i7-6700 and Xeon E5-2430 on all levels of cache display features of LRU-like cache replacement policy in

[9] All data points are averaged from 2 separate runs and each run averages 20 tests. For this data point, one of our tests were contaminated due to the server being shared and used continuously. We have replaced the clearly problematic data with the normal data from the two repetitions.

[10] At this data point, the latency of the Sawtooth is greater than that of the Cyclic, hence the Sawtooth Improvement here should be negative.

Table 1: AMD CPU latency at L1, L2, and L3 in cycles

	Borderline L1 to L2	Borderline L2 to L3	Borderline L3 to Memory
7950X3D avg `for_for`	13.7	45	172.5
7950X3D avg `back_back`	13.7	45	173.2
7950X3D avg Cyclic	13.7	45	172.8
7950X3D avg Sawtooth	9.7	37.7	166.8
7950X3D Sawtooth Improvement	0.29	0.162	**0.03**
7773X avg `for_for`	9.1	40.5	120
7773X avg `back_back`	8.9	32.5	120.5
7773X avg Cyclic	9	36.5	120.25
7773X avg Sawtooth	6.8	28.5	117.5
7773X Sawtooth Improvement	0.25	0.22	**0.02**

Table 2: Intel CPU latency at L1, L2, and L3 in cycles (superscripts are footnote numbers)

	Borderline L1 to L2	Borderline L2 to L3	Borderline L3 to Memory
i7-6700 avg `for_for`	16.5	38.2	276.5
i7-6700 avg `back_back`	15.3	36.9	270.9
i7-6700 avg Cyclic	15.9	37	273.7
i7-6700 avg Sawtooth	12	32	209.2
i7-6700 Sawtooth Improvement	0.33	0.13	0.24
Xeon E5-2430 avg `for_for`	17.16	43.32	171.58
Xeon E5-2430 avg `back_back`	17.1	43[9]	164.8
Xeon E5-2430 avg Cyclic	17.13	43.16	168.19
Xeon E5-2430 avg Sawtooth	14.5	32.73	83.68
Xeon E5-2430 Sawtooth Improvement	0.15	0.24	0.5
Xeon Gold 6230 avg `for_for`	10.9	52.9	116.6
Xeon Gold 6230 avg `back_back`	11.22	52.47	122.1
Xeon Gold 6230 avg Cyclic	11.06	52.69	119.35
Xeon Gold 6230 avg Sawtooth	8.7	35.2	127.5
Xeon Gold 6230 Sawtooth Improvement	0.21	0.33	**0.07**[10]

Figure 8 and Figure 11, respectively. On the other hand, the Xeon Gold 6230 displays LRU-like features in its L1 and L2 level cache, while showing MRU-like feature in its L3 cache.

According to the test results in [6], Intel employs a variant of the MRU replacement policy in many of its CPU architectures [15]. This policy maintains a single status bit for each cache line. When a line is accessed, the respective bit is set to 1. If this was the last bit to be set to 1, all bits for the other lines are set to 0. Upon a cache miss, the leftmost element with a bit set to 0 is replaced.

One thing to be noted is that, for Intel E5-2430, in Figure 13, using the same formula mentioned in Section 3.4, we see a 50% latency reduction by Sawtooth, compared to the Cyclic access latency at 16 MiB, with Cyclic at 168.2 cycles and Sawtooth at 83.7 cycles. The reduction is

$$\frac{|83.7 - 168.2|}{83.7} \approx 0.50$$

The reason of the Xeon E5-2430 Sawtooth result having a greater latency reduction (50% lower) than the L2 results on AMD is that the L3 size is 15 MiB and the next data point larger than L3 size is 16 MiB, which means it has $\frac{15}{16}$ of the total data still being L3 cache hits for Sawtooth and none for Cyclic.

3.5.2 Confirmation with nanoBench. Abel et al. in their work [6] confirmed our belief that the i7-6700 indeed uses an LRU-like cache replacement policy across all three levels of its cache, since i7-6700 shares the same architecture "Skylake" as i7-6500U, which Abel et al. benchmarked for. This provides a valid data point for our benchmark and supports our hypotheses regarding AMD CPUs.

3.6 TLB Effects

In Section 2.6, we have discussed the effect of TLB misses and the need for huge pages. The DTLB sizes for 7773X are 64 entries (L1), 2048 entries (L2), which means it can store up to 2112 pages in total. In other words, the 7773X's DTLB can hold up to 8.25 MiB data without having TLB miss under the default page size of 4 KiB. Thus, once huge page is enabled, we should be able to see an improvement

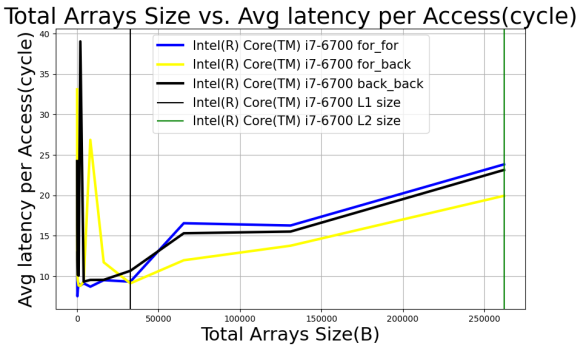

Figure 7: Intel i7 6700 at L1

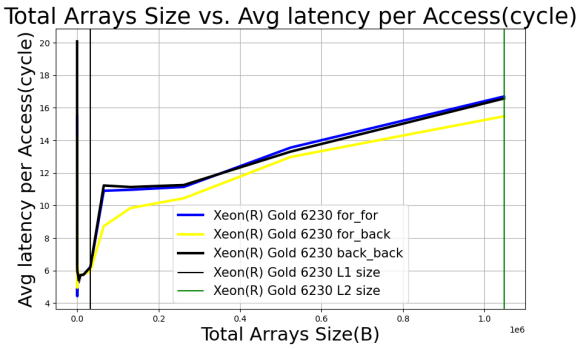

Figure 8: Intel Xeon(R) Gold 6230 at L1

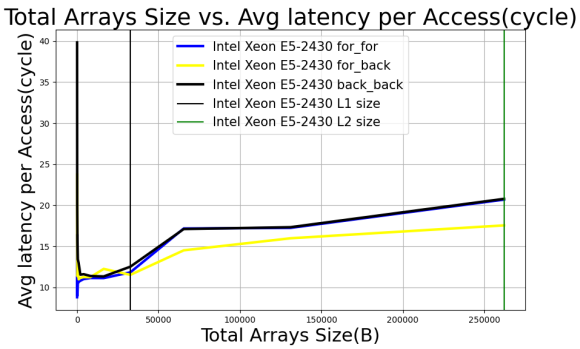

Figure 9: Intel Xeon E5-2430 at L1

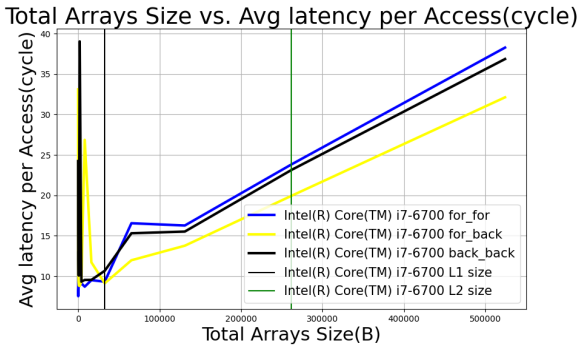

Figure 10: Intel i7 6700 at L2

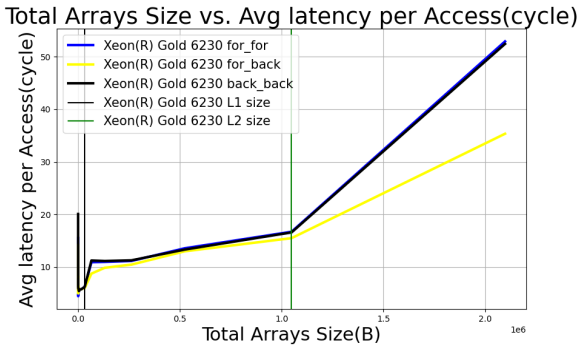

Figure 11: Intel Xeon(R) Gold 6230 at L2

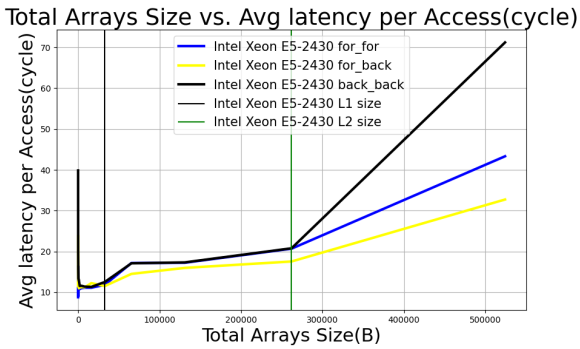

Figure 12: Intel Xeon E5-2430 at L2

on latency for array size that is larger and equal to 16 MiB, which is the next data point larger than 8.25 MiB.

Figure 16 compares the latency on 7773X for the versions of data traversals with regular 4 KiB pages and with 2 MiB pages. We observe the latency starts to differ at data size around 16 MiB and becomes greater as the data size increases beyond that point, which is exactly what we expected.

4 Related Benchmark Tests

In this section, we compare the new test with existing test programs.

4.1 STREAM Benchmark

The STREAM benchmark is a synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for simple vector kernels[14][13]. In this work, we have adopted and modified some part of the STREAM benchmark in order to show the latency cost in cache.

4.2 Core to Core Latency Benchmark

Core to Core Latency Benchmark measures the latency it takes for a CPU to send a message to another CPU via its cache coherence protocol. By pinning two threads on two different CPU cores, it

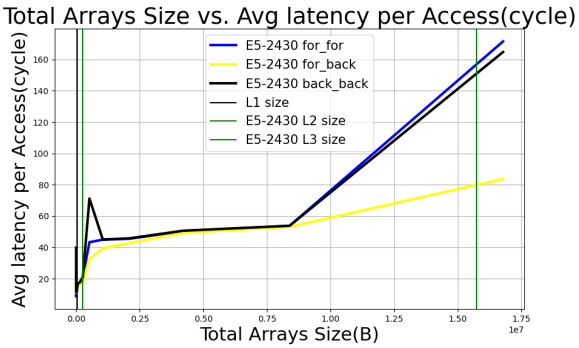

Figure 13: Intel Xeon E5-2430 at L3

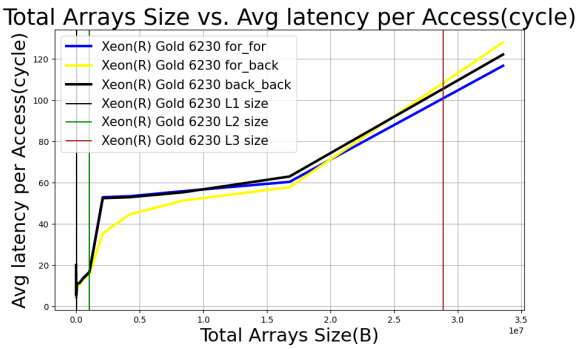

Figure 14: Intel Xeon(R) Gold 6230 at L3

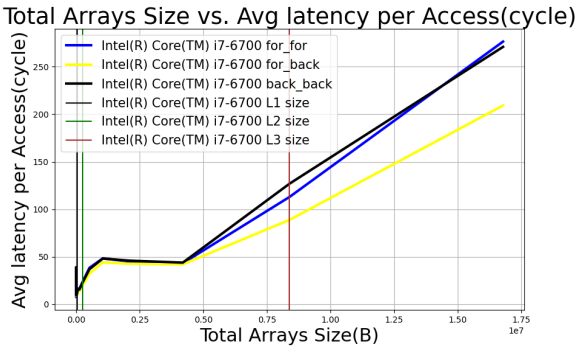

Figure 15: Intel i7 6700 at L3

can enable the compare-exchange operation, and then measure the latency [4].

4.3 Microbenchmarks

Microbenchmark is a benchmark suite that measures performance of CPUs and GPUs [3]. Our test is specifically related their MemoryLatency benchmark. In Microbenchmark, they disable both hardware prefetcher and parallel loading by using dependent loads. We use the same technique. However, their result shown in Figure 17

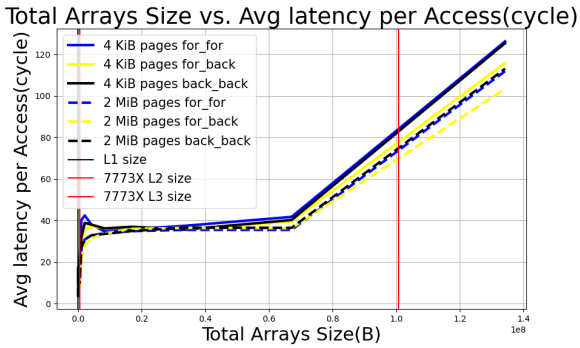

Figure 16: EPYC 7773X Latency with 4 KiB pages and with 2 MiB pages

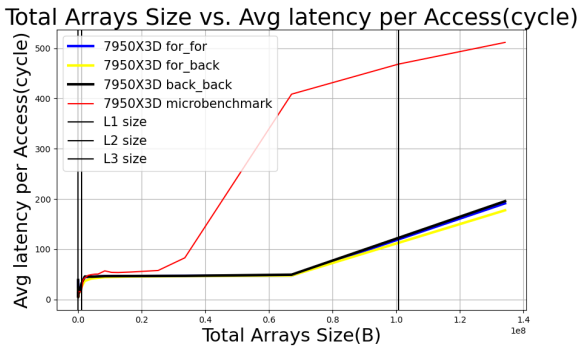

Figure 17: R9 7950X3D Microbenchmark with SiC Benchmark comparison on 2 MiB pages[11]

seems to be disagreeing with our benchmark, tested on R9 7940X3D. One possible explanation is that, in Microbenchmark, they did not account for the asymmetric CPUs on the machine and bound the benchmark process to a core that is on 7950X3D's CCD1, which has only 32 MiB of L3 cache due to the lack of V-Cache. Thus, we observe the latency peaks at 64 MiB instead of 128 MiB. Another possible reason is Microbenchmark uses `gettimeofday()` instead of more precise timers such as `rdtsc()` or `rdtscp()`, which is used by our test.

4.4 Intel Memory Latency Checker

Intel® Memory Latency Checker (Intel® MLC) is a tool used to measure memory latencies and bandwidth, and how they change with the increasing load on the system [16]. Intel® MLC can disable the hardware prefetcher, either by controlling MSR or enabling random accesses. In Figure 18, Intel® MLC seems to have measured an off-chip memory latency of 107 cycles for data size of 128 MiB, which conflicts with our common understanding of the off-chip memory latency. In a later test on cache to cache (L2 to L2) latency test, Intel® MLC reported that from a local socket whether the

[11] Intel® MLC requires to be ran on 2 MiB pages. In order to keep the results consistent, SiC, Microbenchmark and Intel® MLC are all ran on 2 MiB pages.

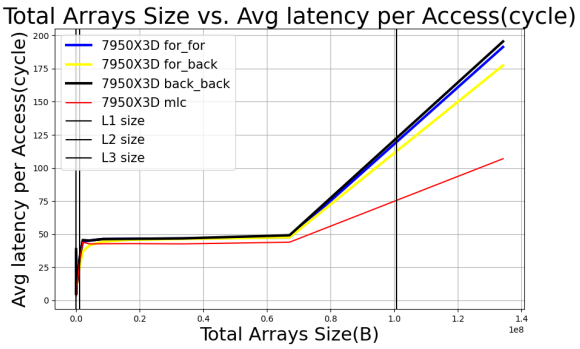

Figure 18: EPYC 7773X Memeory Latency Checker with SiC Benchmark comparison on 2 MiB pages[5]

data transferring from L2 to L2 is a hit or not, the latency is always 108.3 cycles, which means that Intel® MLC could have some problems with its L2 latency measurement that led to the incorrect measurement of the off-chip memory latency.

4.5 nanoBench

nanoBench detects replacement policies used on Intel machines, including the set specific parameters used by insertion-based policies [6]. Their paper shows results for 13 Intel machines. nanoBench uses generated kernel code and hardware counters, which provides in-depth measures on Intel machines. It shows that upper level caches use LRU. Our results on the Intel machine agree with theirs. Unlike SiC, nanoBench has not been ported to AMD machines. Since SiC is less complex and does not require kernel access (except for setting huge pages), it is more portable. We have analyzed AMD machines with LLC of size 96MB (per CCD) and an Intel machine with 27.5MiB LLC. The maximal size of LLC analyzed in the previous work is 8MB.

4.6 Pointer Chasing Benchmarks

Both Memory Stress Benchmark and Google Multichase Benchmark utilize the pointer chasing method to avoid prefetching and thus retaining a relatively accurate measurement of the machine [11] [5]. SiC Benchmark uses a similar dependent load design to avoid prefetching.

4.6.1 Memory Stress Benchmark. Memory Stress (Mess) framework provides an unified view of the memory system benchmarking, simulation and application profiling [11]. Memory Stress Benchmark has a similar design comparing with our SiC benchmark. It utilizes the conventional "pointer chasing" method, while SiC Benchmark uses a similar dependent load design. In pointer chasing, the next address depends on the pointer value stored in the current address. In dependent loads, the next address is the value stored in the current address with the triangular sequence modification, whose access pattern is not contiguous.

Conventional pointer chasing method does not involve the triangular sequence part, which we use this feature to avoid possible aggressive prefetching. i.e. in pointer chase, the successive memory location being accessed may be contiguous from the current location, and the access latency therefore reduced by prefetching. This cannot happen by accident in SiC due to the extra triangular sequencing.

4.6.2 Google Multichase Benchmark. The Google Multichase Benchmark is a performance testing tool designed to measure memory latency, bandwidth, and loaded-latency across different array sizes and thread configurations, using pointer chasing method [5].

However, their result, tested on EPYC 7773X, varies from run to run. One possible reason is that in Multichase, although the total memory is allocated contiguously in an arena, the benchmark permutes the order of objects. Therefore, the distances among interconnected objects varies on each run. This can potentially disturbs the results.

5 Conclusion

In summary, this work introduces a new cache latency benchmark that employs triangular accesses in defeating hardware prefetcher and deploys Sawtooth and Cyclic data traversal to measure the cache replacement policy of the benching processor. We have measured the new 3D V-Cache technology in AMD's CPU and report that the cache is partitioned rather than shared, and the cache replacement policy is more similar to random eviction than it is to LRU. Finally, we have also compared our findings with other existing latency benchmarks.

Modern microarchitecture implementations introduce sophisticated optimizations that hide latency due to cache misses, including pipelining, prefetching, and memory parallelism. We have developed a new test borrowing the idea of triangular accesses and extending it to use dependent loads and have two directions of traversal with introducing additional memory accesses.

Acknowledgments

We would like to express our sincere gratitude to Mingzhe Du for her suggestions and for Michael Scott for the guidance and support throughout this project. We would also like to thank all the anonymous reviewers of MEMSYS'24 for their constructive feedback.

References

[1] 2007. sparsehash. https://github.com/sparsehash/sparsehash.
[2] 2015. Triangular numbers mod 2^n. https://fgiesen.wordpress.com/2015/02/22/triangular-numbers-mod-2n/
[3] 2021. Microbenchmarks. https://github.com/clamchowder/Microbenchmarks.
[4] 2022. core-to-core-latency. https://github.com/nviennot/core-to-core-latency.
[5] 2024. multichase. https://github.com/google/multichase.
[6] Andreas Abel and Jan Reineke. 2020. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. http://arxiv.org/abs/1911.03282
[7] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. [n. d.]. Swiss Tables Design Notes. https://abseil.io/about/design/swisstables
[8] Erik Berg and Erik Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Austin, Texas). 20–27.
[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. MIT press.
[10] Jack J. Dongarra. 2022. The evolution of mathematical software. *Commun. ACM* 65, 12 (2022), 66–72. https://doi.org/10.1145/3554977
[11] Pouya Esmaili-Dokht, Francesco Sgherzi, Valeria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, German Llort, Petar Radojkovic, Miquel Moreto, Judit Gimenez, Xavier Martorell, Eduard

Ayguade, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. 2024. A Mess of Memory System Benchmarking, Simulation and Application Profiling. arXiv:2405.10170 [cs.AR] https://arxiv.org/abs/2405.10170

[12] Linux Foundation. [n. d.]. Linux Preemption Models Wiki. https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models Accessed on Mar/22/2024.

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

[14] 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.

[15] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Systems* 37, 2 (November 2007), 99–122. https://doi.org/10.1007/s11241-007-9032-3

[16] Vish Viswanathan, Karthik Kumar, Thomas Willhalm, Sri Sakthivelu, and Sharanyan Srikanthan. [n. d.]. Intel® memory latency Checker v3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[17] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2010. Online cache modeling for commodity multicore processors. *Operating Systems Review* 44, 4 (2010), 19–29.

[18] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (mar 1995), 20–24. https://doi.org/10.1145/216585.216588

[19] Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Transactions on Architecture and Code Optimization* 14, 4, 34:1–34:26. https://doi.org/10.1145/3134437