Dissecting Scale-Out Applications Performance on Diverse TLB Designs

Tianmai Deng, Yixiao Xu, Chi Zhang, Zhengwei Qi*
Shanghai Jiao Tong University
Shanghai, China
{tianmaideng, xuyixiao.2019, sjtu-zhangchi, qizhwei}@sjtu.edu.cn

***Abstract*—Scale-out applications, such as various big data systems and memory computing programs comprise an important software stack in clouds. Such applications usually have large memory data footprint as well as code sizes, thus stressing the CPU's TLB efficiency. In this paper, we experimentally evaluate how various TLB design choices in modern off-the-shelf x86 CPUs impact the performance of scale-out applications. The findings aim to guide the partitioning schemes and capacity planning of TLBs, and software-hardware co-design for emerging applications.**

***Keywords*—performance evaluation, scale-out applications, TLB structures, MMU overhead, TLB miss rates**

I. Introduction

In cloud service environments, dominant scale-out applications [1], such as Hadoop and Spark for distributed data analysis, and back-end applications like Nginx and MySQL for web services, are widely deployed. These applications exhibit markedly different memory characteristics at runtime compared to traditional Linux applications. In particular, the executable binary file size and the data working set size [2] of scale-out applications have significantly increased. This poses unprecedented memory access pressure [3] on non-supercomputing data center machines.

At the same time, as the main frequency increases and pipeline efficiency improves [4], CPUs favor continuous feeding of instructions to maximize their processing power. However, modern scale-out applications tend to issue a large body of load/store operations that stress the MMU for address translation. Address translation in modern 64-bit CPUs is non-trivial, which contains 4-layer page walks along with cache misses for PTE accesses. Thus, TLBs are indispensable components to mitigate such expensive memory operations.

For high-performance scenarios on modern CPUs, TLB entries should be filled only once (initial translation) and held for the entire application lifecycle. In such cases, all memory accesses within the application avoid address translations and exhibit the same TLB access latency, which is fast due to the hardware nature of TLB. However, in real-world scenarios, TLB misses still happen [5] and damage application performance due to bad data locality, code and data contention of TLB entries, or fragmented address accesses.

Previous works conducted performance analysis on data center applications, such as middleware [6] and serverless frameworks [7]. Especially, Michael et al. [5] analyzed the MMU overhead for scale-out applications, which can account for 1.8% to 16% of its runtime. However, such overhead is by far not well understood with regard to the specific TLB designs in modern CPUs. This paper aims to fill this gap and advance the understanding.

To investigate the impact of TLB design on scale-out applications performance, this paper conducts experiments on different processors, including Intel's 9th and 12th generation processors and the AMD Zen3 processor, which use the same x86 ISA but embody very different TLB designs. We analyze the performance of applications and measure the miss rate of TLB, under various workloads and system huge page configurations. Specifically, we explore various TLB design details and clarify how different TLB capacities and structures affect the scale-out application performance.

The main contributions of this paper include:

- We find that some scale-out applications still suffer from a non-negligible data memory performance overhead of up to 10.9% due to TLB misses.
- Applications with small hot code areas (such as data-caching, in-memory-analytics, and graph-analytics) have reduced code performance loss to less than 0.1% on the latest CPUs.
- Applications with store-major memory operations (such as memcached and redis) tend to have better performance when the underlying CPU has a unified dTLB for loads and stores.
- Applications with frequent accesses to a small part of hot data will be greatly impacted by the L1 dTLB latency. Thus, an L1 dTLB with higher associativity (thus lower latency) combined with extensive uses of huge pages (manual application code tuning or OS-level allocator policy) can effectively improve their performance.

The rest of the paper is organized as follows. Section 2 introduces the potential performance impact of paging and TLB design on scale-out applications. Section 3 presents our experimental setup and evaluation methodology. Section 4 discusses the results of our experiments, and presents the performance analysis and discussion. Finally, Section 5 concludes the paper.

DOI reference number: 10.18293/SEKE23-139

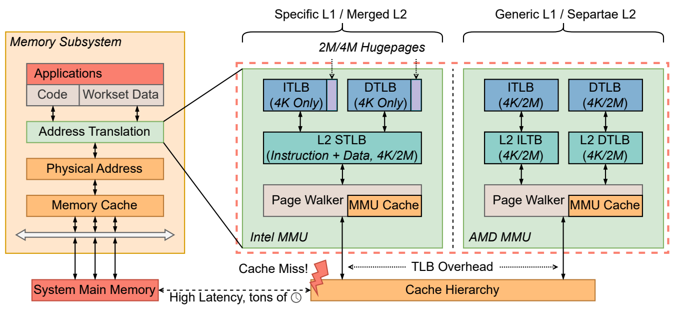

Fig. 1. Memory subsystem and MMU structures in different processors. *The TLB entries for 1G huge pages are omitted, and in the latest Intel microarchitecture, dTLB is divided into store TLB and LOAD TLB [8], which is simplified here.

II. Performance impact of TLB

This section illustrates that the paging memory system is not suitable for scale-out applications, and describes several TLB designs and their impact on the performance of different applications.

A. *TLB, the legacy of paging memory*

The Memory Management Unit enables efficient memory allocation and management through paging. In the early days of the computer, the need for memory performance in applications was not yet as great as the demand for memory capacity [9], so the more flexible and efficient memory utilization that paging brings outweighs the additional access latency when an address translation occurs. Nowadays, 64-bit CPUs are widely used where 4-layer page tables replace the 2-layer tables on 32-bit CPUs and incur a 2x latency penalty. Also, the main frequency of CPUs has greatly increased over time. The CPU has to wait for more cycles when memory access stalls due to address translation.

However, scale-out applications are placing higher demands on memory capacity [3], and the additional logic introduced is causing the binary file size to balloon [2]. In addition, the working set size has grown geometrically to handle the ever-increasing amount of data. The scale-out applications demand far more performance and capacity from both code and data memory than they did when paging was proposed, and the paging design with flexibility is no longer performant for scale-out applications.

TLB [9] and huge page [10], as mitigants of the paging memory system, are unable to fully resolve the divergence between the scale-out applications and the paging memory system. Every TLB miss can result in applications being exposed to the paging memory system and having to endure meaningless page walk latency [11]. Therefore, how to utilize TLB resources as much as possible on top of the decaying paging memory system to avoid negative performance impact on scale-out applications is a matter of concern.

B. *Impact of TLB Structures*

Different TLB designs are not oriented to the same memory access pattern and may exhibit different performance and utilization for different applications and huge page setups.

TABLE I
TLB Specifications (all sizes = 4K/2M/1G)

Intel Coffee Lake (2017) [8]			
iTLB	4K	128 entries	512K
	2M	8 entries	16M
dTLB	4K	64 entries	256K
	2M	32 entries	64M
	1G	4 entries	4G
sTLB	4K/2M	1536 entries	6M/3G
	1G	16 entries	16G
Intel Golden Cove (2021) [8]			
iTLB	4K	256 entries	1M
	2M	32 entries	64M
dTLB (store)	all sizes	16 entries	64K/32M/16G
dTLB (load)	4K	64 entries	256K
	2M	32 entries	64M
	1G	8 entries	8G
sTLB	4K/2M	1024 entries	4M/2G
	4K/1G	1024 entries	4M/1T
AMD Zen3 (2020) [12]			
L1 iTLB	all sizes	64 entries	256K/128M/64G
L1 dTLB	all sizes	64 entries	256K/128M/64G
L2 iTLB	4K/2M	512 entries	2M/1G
L2 dTLB	4K/2M	2048 entries	8M/4G
	1G	64 entries	64G

In the L1 TLB, AMD and Intel adopt two different implementations of TLB entry for various page sizes, which we call generic entry and specific entry, as illustrated in Figure 1. Generic entry design [8] is that each L1 TLB entry can store address translation results for all page sizes. However, this design limits the number of TLB entries due to the need for recording more bits per entry. In contrast, the specific entry design [12] stores 4K pages and 2M huge pages in separate entries in the L1 TLB. However, the number of entries reserved for huge pages is limited, which reduces the effective range of address mapping when using huge pages (Table 1).

There are also two different approaches for L2 TLB. Intel's approach [8] is combining iTLB and dTLB in the merged L2 TLB, also known as the sTLB, to record address translation results for both code and data memory. Although TLB entry updates may face contention, this design allows for more flexible capacity sharing. AMD chooses to separate the sTLB into distinct L2 iTLB and L2 dTLB [12] to store the address translation results for code and data memory, respectively, without interference. However, this approach may not fully utilize all hardware resources when the pressure of the two types of memory is unbalanced.

III. Methodology

This section describes the test load selection, test environment setup, and test methods used in our experiments.

A. *Simulated Load*

CloudSuite benchmark [13] is a set of test applications or collections for different scenarios that cover various situations that may occur in data centers. To measure the performance of scale-out applications, we used CloudSuite benchmark 4.0 as a simulated data center workload in our study.

B. Test Environment Setup

To compare the performance impact of different TLB capacities and structures, we set up the experimental environment on three hosts with different processors:

- Host A: 6-core Intel Core i5-9600K (Coffee Lake) CPU running at 3.70GHz with DDR4 32G memory.
- Host B: 8-core Intel Core i7-12700 (Golden Cove) CPU running at 4.50GHz with DDR4 16G memory.
- Host C: 8-core AMD Ryzen 5700G (Zen3) CPU running at 4.60GHz with DDR4 32G memory.

The detailed TLB specifications are recorded in Table 1. All hosts run the Debian 11.6 system with Linux kernel 6.0.12-1.

C. Performance Event

To analyze the running characteristics of the same workloads on processors with different TLB capacities and structures, we bind client and server applications to different cores, while only measuring the cores running the server application. To collect processor performance events during testing, we employed the `Linux perf` tool, and the average of three test results was computed.

D. Huge Page

If only 4K pages are used, the TLB can provide an extremely limited range [14] of address mapping (Table 1), which is much smaller than the size of the scale-out application's working set, or even the size of its code memory, but the use of huge pages can effectively alleviate this limitation.

In Linux, there are two ways to use huge pages: Transparent Huge Pages (THP) and hugetlbfs. THP is transparent to the application and does not require modification of the application, while hugetlbfs is not. And starting from kernel 5.14, THP can not only be used for data but also executable memory, mapping application code areas with huge pages. However, if the memory pages are not aligned to 2MB or not contiguous, the promotion rate of THP will not only decrease further but also introduce noticeable memory bloat [15].

IV. Experiment Result and Analysis

In this section, we analyze application performance overhead on processors with various TLB designs. Then, we investigate the effects of using huge pages. Additionally, we analyze the importance of L1 iTLB and L1 dTLB for applications with limited hot code areas and frequent hot data accesses. Furthermore, we explored the impact of a separate store dTLB on data memory performance. Finally, we measure the effective hit rate of L2 TLB on different structures and whether it matters to applications.

A. Is MMU overhead still significant for scale-out applications on the latest x86 processors?

In this section, we measure and calculate the proportion of cycles consumed by page walks (Host A and Host B) or pipeline stalled cycles (Host C) during application runtime to evaluate the MMU overhead caused by TLB misses for different scale-out applications. Figure 2 shows the results.

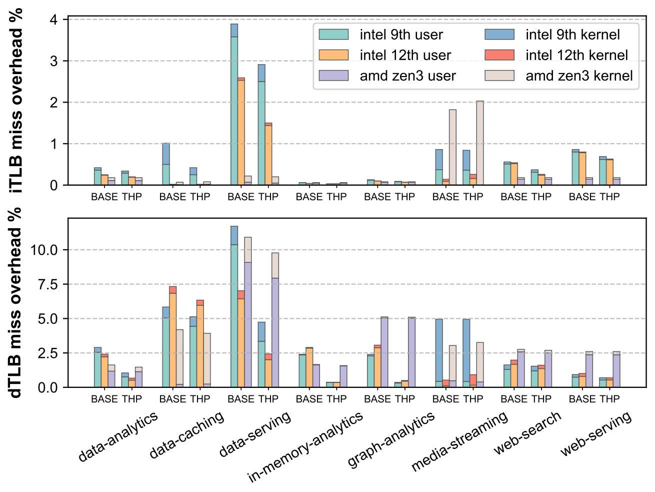

Fig. 2. Intel: Percentage of cycles consumed by page walks relative to the total cycles. AMD: Percentage of cycles caused by TLB misses leading to front-end/back-end stalling.

For most scale-out applications, the performance overhead due to iTLB misses is not significant. The average iTLB misses overhead is only 0.98%, 0.56%, and 0.35% on Host A, Host B, and Host C. However, for applications with large hot code areas such as data-serving and web-serving, the iTLB miss overhead is still non-negligible, especially on processors with small TLB capacities such as Host A, where the overheads are 3.89% and 0.86%, respectively.

Since the MMU overheads are not comparable due to different performance event definitions between Host C and the others, we analyze the TLB miss rates (Figure 3 and Figure 4). We find that the L2 iTLB miss for most applications is less than 0.25 MPKI, for applications with large hot code areas it still does not exceed 1 MPKI. However, the L2 iTLB miss rate of these scale-out applications on Host C is on average 63.26% higher than that on Host B, which suggests that applications running on Host C may get worse code performance.

Moreover, by comparing the code performance overhead on Host A and Host B, we find that Host B has indeed alleviated the code memory address translation overhead by increasing the iTLB and sTLB capacity. The code performance overhead on average decreases by 39.61%, and in data-caching the overhead remarkably decreases by 98.2%.

In contrast to the low code performance overhead, our results show that most scale-out applications still suffer from significant data memory performance overhead due to dTLB misses. Specifically, these applications have an average overhead of more than 3.68% on Host A and Host B, and applications with higher data memory pressure, such as data-caching and data-serving, introduce up to 11.71% and 7.33% overheads.

Findings:

- In scale-out applications, the overhead caused by iTLB misses is not significant, with most applications showing a code performance loss of less than 1%.

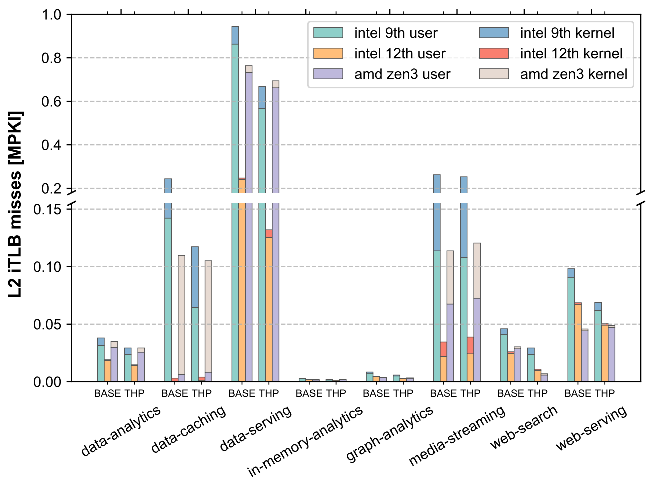

Fig. 3. L2 iTLB miss rate, measured as the number of misses per thousand instructions (MPKI).

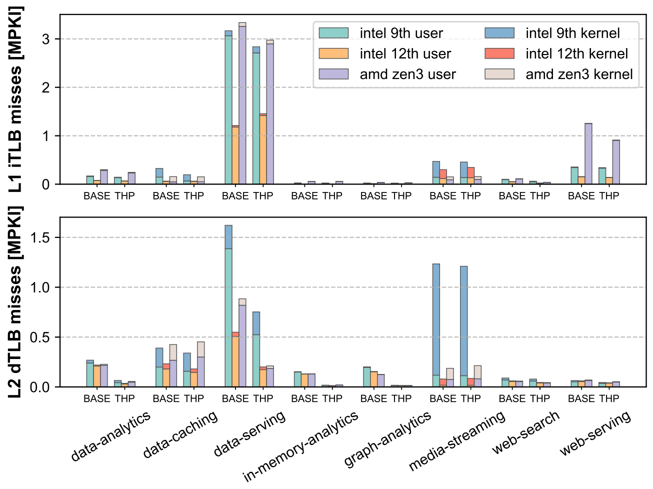

Fig. 4. L1 iTLB and L2 dTLB miss rate, measured as the number of misses per thousand instructions (MPKI).

- Most scale-out applications still suffer from a non-negligible data memory performance overhead. The dTLB miss overhead on the data-serving application exceeds 10% on older processors, which can be reduced by over 25% by enlarging dTLB capacity.
- In database applications and others with large hot code areas, the iTLB miss overhead is still noticeable and should be given due attention.
- Separating the L2 iTLB from the L2 TLB has a non-efficient improvement on the overall application code performance.

B. Can huge pages improve the code and data memory performance of applications?

The comparison of MMU overhead before and after enabling THP reveals that using huge pages for code and data memory leads to better overall performance, resulting in an average of 12.16% reduction in iTLB miss overhead and 27.43% in dTLB miss overhead. For data-serving applications, enabling THP leads to a significant reduction in iTLB miss overhead by an average of 23.30%, resulting in better code performance. Additionally, for data-analytics, in-memory-analytics, and graph-analytics applications, using huge pages leads to a reduction in data memory performance overhead by 48.49%, 58.73%, and 56.45%, respectively. Even for web-serving databases that are not suitable for huge pages [16], enabling THP still reduces iTLB and dTLB miss overhead by 12.58% and 18.41%. However, it is important to consider the potential long-tail in memory access latency due to THP memory page locking [15], and explicit use of huge pages is recommended instead.

Findings:

- In most scale-out applications, even those with sparse data memory access patterns like database applications, enabling huge pages has a significant improvement in both code and data memory performance.

C. What can be observed when running on processors with different TLB designs?

We analyze how different TLB designs affect application performance. Figure 4 illustrates the L1 iTLB and L2 dTLB miss rate on different processors. For every application, Host B outperforms the others, with an average 65.28% and 59.47% reduction in iTLB miss rate to Host C and Host A.

In data-caching, in-memory-analytics, graph-analytics, and web-search applications, Host B has a lower L2 dTLB miss rate than Host A, but with a higher data memory performance overhead (Figure 2). One possible cause of this phenomenon is that the Linux kernel swaps out [17] some pages to disks due to limited system memory capacity, which causes orders of magnitude data memory performance degradation of the program when accessing these pages.

For code performance, except for applications with limited hot code areas, such as data-caching and media-streaming, Host C's fully associative design of L1 iTLB is not as effective as larger but less associative L1 iTLB on other applications. The L1 iTLB miss rate of Host C is 14.33% higher compared to Host A. Host B continues to increase its capacity while keeping its associativity unchanged, further reducing the average L1 iTLB miss rate by 59.47%. This is particularly evident in data-caching, an application with limited hot code areas, where the L1 iTLB miss rate of Host B is reduced by 63.67%.

Regarding the huge pages, we found that enabling THP resulted in reduced iTLB and dTLB miss rates for all three processors in most applications, with an average reduction of 14.21% and 46.72%, respectively. And the data in Figure 3 shows that enabling THP also led to a reduced average L2 iTLB miss rate of 22.52%.

Lastly, we observed an abnormal increase in the L1 iTLB miss rate of the data-serving application on Host B when enabling THP, with an increase of 20.03%. However, a similar phenomenon does not occur on Host A, which has fewer iTLB

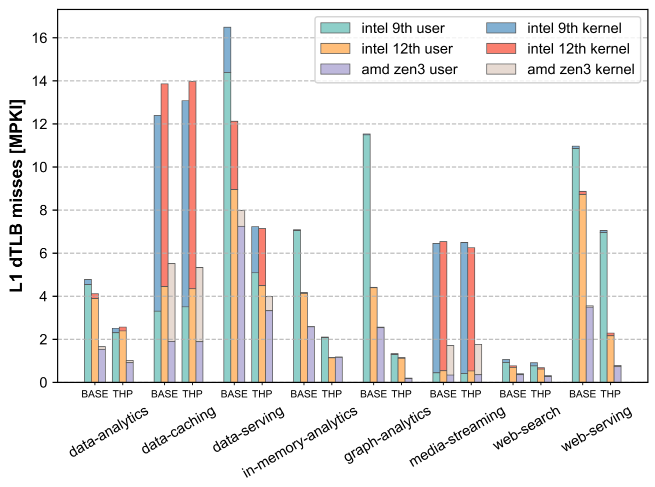

Fig. 5. L1 dTLB, for different processors, measured as the number of misses per thousand instructions (MPKI).

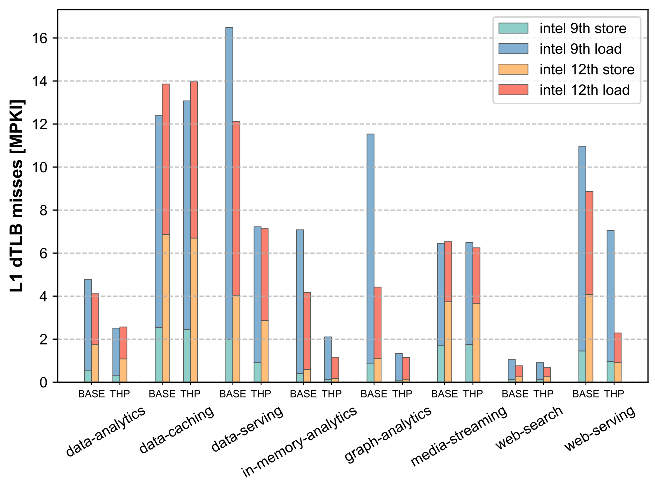

Fig. 6. L1 dTLB, for Load and store operations, measured as the number of misses per thousand instructions (MPKI).

entries for huge pages. This may be attributed to the iTLB of Host B has a huge capacity for 4K pages and can perfectly fit the hot code areas of the application, while not limited by the limited number of huge page entries.

Findings:

- No matter which processor and what TLB design is considered, enabling huge pages can lead to an overall improvement in application performance.
- For scale-out applications with limited hot code areas, the impact of L1 iTLB capacity has reached a critical point. Running on processors with larger L1 iTLB can almost eliminate the address translation overhead on code performance.

D. How does the design of L1 dTLB affect centralized hot data memory accesses?

Centralized memory accesses to a small part of hot data will be greatly impacted by the L1 dTLB design. In VM-based cloud environments, nested page tables make the address translation overhead even higher [18]. For applications with frequent accesses to hot data that may be sensitive to latency, it is necessary to ensure that all memory access hit the TLB to avoid data memory address translation.

Figure 5 shows the miss rates of L1 dTLB. For all scale-out applications, it can be observed that Host C's L1 dTLB performs significantly better than Host A and Host B. The average L1 dTLB miss rate for Host C is only 3.2487 MPKI, which is 47.39% and 36.73% lower than Host A and Host B, respectively. Given the similarity of the L1 dTLB capacity among the three hosts, Host C's fully associative design is likely the reason for its higher hit rate.

After enabling THP huge pages, the L1 dTLB miss rates on Host C, Host A, and Host B are reduced for any applications. The average reduction is up to 58.52%, indicating that using huge pages is also extremely useful for reducing the hot data memory access latency.

Findings:

- For applications requiring low latency access to hot data, a processor with higher associativity on L1 dTLB should be considered.
- The use of huge pages effectively reduces the L1 dTLB miss on different processors, which further reduces access latency on hot data access.

E. Do hybrid load/store applications suffer a performance hit due to store dTLB?

We noticed that in Intel's latest dTLB design, the store operations have been allocated a small separate area. Therefore, we separately measured the hit rates of load and store accesses in the L1 dTLB, as shown in Figure 6.

Firstly, for all applications running on Host B, the miss rate of the store operation in L1 dTLB is significantly higher than that on Host A, with an average increase of 129.73%. However, by avoiding store operations from competing with load operations for limited TLB resources and mitigating the interference with the address translation of load operations, the TLB miss rate of load operations is dramatically reduced by 44.57%. This further contributes to a 22.22% decrease in the overall miss rate of the L1 dTLB, thus providing better data memory performance for applications.

However, for applications with store-major memory operations, such as data-caching, the store dTLB miss rate increased by 187.55%, despite a decrease of 29.27% in the load L1 dTLB miss rate. The high store miss rate leads to overall performance degradation of the L1 dTLB. Nonetheless, since most applications mainly issue load operations, the separate store dTLB does help reduce the dTLB miss rate without noticeably increasing the dTLB capacity.

Findings:

- For applications that have a high proportion of store operations, running on processors without separated dTLBs can probably provide better overall performance.

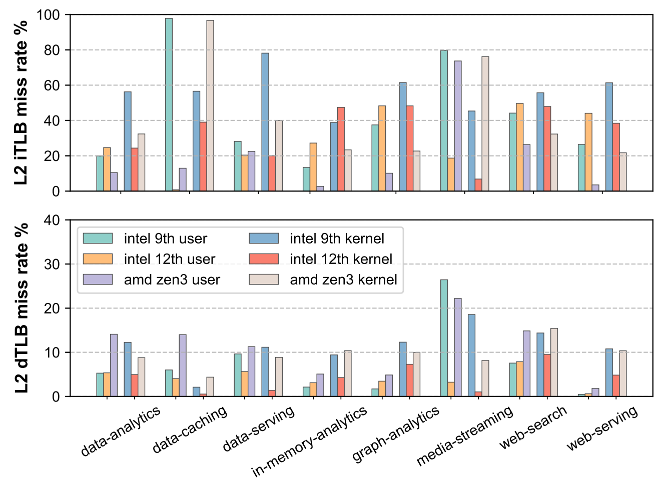

Fig. 7. The percentage of L2 iTLB and L2 dTLB miss counts to the L2 iTLB and L2 dTLB access counts

F. Does the separated L2 TLB matter for scale-out applications?

We define L2 TLB self-miss rate (Figure 7) as the number of L2 TLB misses over the number of L2 TLB accesses.

Benefiting from the separate design, Host C exhibits lower L2 iTLB self-miss rates and provides better code performance for most applications at the same L1 iTLB efficiency, except for the data-caching and media-streaming, where Host B performs exceptionally well due to its huge L1 iTLB responds to most address translation requests.

For data memory performance, Host C has even worse L2 dTLB self-miss rates despite separating the L2 dTLB from L2 TLB with dedicated 2048 entries. Thus, some applications' data memory performance on Host C may not be as good as that of Host B, which has a smaller total L2 TLB size but also a lower L2 dTLB miss rate.

Findings:

- For scale-out applications, the separated L2 TLB can slightly improve the code performance, but not the data memory performance.

V. Conclusion

In this paper, we measured and analyzed the performance overhead suffered by a variety of scale-out applications due to the paging memory system and address translation. We found that even though the latest x86 processors try to reduce the MMU overhead on scale-out applications by increasing capacity and associativity, they still cannot fully mitigate the degradation of code and data memory performance due to TLB misses. When running scale-out applications, selecting a processor with the appropriate TLB design can maximize performance based on the memory access characteristics and requirements, such as frequent hot data accesses or mixed loads and stores. Noteworthy, we found that for applications with limited hot code areas, the current L1 iTLB capacity has reached a critical point where almost all code address translations overhead can be eliminated. Finally, we deepened the understanding of the MMU overhead suffered by scale-out applications due to paging by incorporating in-depth analysis on TLB design, and we hope the findings can guide the planning of TLBs and CPUs, and software-hardware co-design for the next generation of applications.

VI. Acknowledgement

This work was supported in part by National NSF of China (No. 62141218), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Morgan & Claypool Publishers, 2018.

[2] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, Mar. 2012, pp. 37–48.

[3] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *The 40th Annual International Symposium on Computer Architecture, ISCA*, Jun. 2013, pp. 237–248.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012.

[5] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. M. Swift, "Performance analysis of the memory management unit under scale-out workloads," in *2014 IEEE International Symposium on Workload Characterization, IISWC*, Oct. 2014, pp. 1–12.

[6] C. Röck, S. Harrer, and G. Wirtz, "Performance benchmarking of BPEL engines: A comparison framework, status quo evaluation and challenges," in *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE*, Jul. 2014, pp. 31–34.

[7] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Analyzing open-source serverless platforms: Characteristics and performance (S)," in *The 33rd International Conference on Software Engineering and Knowledge Engineering, SEKE*, Jul. 2021, pp. 15–20.

[8] Intel Corporation, "Intel® 64 and ia-32 architectures optimization reference manual," Feb. 2023, iD. 671488.

[9] B. L. Jacob and T. N. Mudge, "A look at several memory management units, tlb-refill mechanisms, and page table organizations," in *ASPLOS*, Oct. 1998, pp. 295–306.

[10] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *54th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, Oct. 2021, pp. 1106–1120.

[11] A. Bhattacharjee, "Large-reach memory management unit caches," in *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, Dec. 2013, pp. 383–394.

[12] AMD Corporation, "Software optimization guide for amd family 19h processors (pub)," Jun. 2020, iD. 56665, Revision 3.0.

[13] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, Apr. 2016, pp. 122–132.

[14] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *ASPLOS*, Apr. 2017, pp. 435–448.

[15] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *ASPLOS*, Mar. 2018, pp. 679–692.

[16] T. Michailidis, A. Delis, and M. Roussopoulos, "MEGA: overcoming traditional problems with OS huge page management," in *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR*, Jun. 2019, pp. 121–131.

[17] T. Kay, "Linux swap space," *Linux Journal*, vol. 2011, no. 201, p. 5, 2011.

[18] S. K. Barker and P. J. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems, MMSys*, Feb. 2010, pp. 35–46.