Cache Latency Analysis for Processing-In-Memory Applications

Leon Riesebos
Department of Electrical and Computer Engineering
Duke University
Email: leon.riesebos@duke.edu

Amber Johnson
Department of Electrical and Computer Engineering
Duke University
Email: amber.j.johnson@duke.edu

I. Introduction

This paper is the final report for the course ECE 590-05, Conventional and Emerging Memory Systems. For our work, we conducted a project in the area of Processing-In-Memory (PIM) technologies. PIM technologies can be applied to various memory technologies, including emerging ones [1]–[4], and cover various application domains, including neural networks [5]–[7].

To make sure our project fits in the provided time budget, we decided to keep the project focused on topics related to PIM technologies applied to commodity SRAM in the context of general-purpose HPC. The performance of PIM technologies is dependent on the performance of the used memory. Especially for computing-in-cache related techniques [8], [9], cache latencies play an important role. For our project, we decided to perform cache latency analysis on various machines to gain a better conceptual understanding of how cache latency can influence the performance of in-cache computing technologies.

In Section II we provide a brief motivation for our project which is followed by the project scope in Section III. In Section IV we will introduce the methods that we will use for the remainder of this work. The results of our measurements and the analysis of that data are presented in Section V. Our future work is listed in Section VI and this paper is concluded in Section VII

II. Motivation

For the last decades, the computational power of our systems has been limited by a phenomena known as the *memory wall*. The memory wall refers to the fact that the computational power in our systems has increased faster than the performance of our memory systems, leading to a performance mismatch. Various techniques have been introduced to alleviate the memory wall of which the most common one is CPU caches. Caches introduce a memory hierarchy that can exploit temporal and data locality, thereby hiding the limited performance of the main memory. Modern CPU's rely heavily on their cache systems and devote a significant fraction of die area (40-60%) to such systems. While cache systems improve overall performance, the fundamental problem of relatively slow main memory performance compared to computational performance remains.

Processing-In-Memory (PIM) has been proposed as a potential solution to break the memory wall. By moving computation near or preferably in memory, the gap between computing and storage is reduced, potentially breaking fundamental barriers and increasing performance significantly. The performance potential of PIM technologies is caused by the availability of massive data-parallelism in combination with reduced data movement. Early research into PIM [10] focused on adding computational components to existing memory structures, which is known today as Near-Memory Computing (NMC). More recent architectures based on *bit-line computing* [11], [12] enable computing capabilities in existing SRAM structures with minimal area overhead. Hence, researchers focused on in-cache computation architectures for data-parallel applications [8], [9]. Such in-cache computation architectures could potentially transform large on-chip caches into massive data-parallel accelerators. Bit-line computing provides a limited set of parallel bitwise operations. With limited area overhead, hardware for a *bit-serial* computation scheme can be added, which enables the device to perform various arithmetic operations too.

With potentially much more responsibility in the cache hierarchy of our systems, their performance and latency become more important. In-cache computation architectures are expected to be highly data-parallel and throughput of such technologies would probably not be their first limitation. A potential performance bottleneck of in-cache computing systems could be the access latency. CPU's can access their registers in a single clock cycle while cache latencies, especially for larger levels, tend to be in the order of tens of cycles.

III. Project scope

We decided to design a project in the context of PIM and in-cache computing technologies that is feasible in the time budget available to us. As discussed in Section II, cache latencies could play an important role in the performance of future in-cache computational systems. Hence, we propose to characterize cache latencies of systems today to understand better how the properties of current cache technologies could influence the performance of future in-cache computational accelerators.

In our project we will characterize the cache hierarchy in systems available to us. Our main focus will be cache latency

of various levels in the memory hierarchy. We will mainly be focused on the cache hierarchy of CPU's since those are potential candidates for early in-cache computing accelerators. Our quantitative analysis should provide us insight into the latency metrics of different systems.

The second part of the project is to provide an analysis to understand how in-cache computation architectures, as proposed in [8], [9], could perform on systems existing today. This part of the project will be mainly analytical and potentially based on the earlier obtained characterization data.

Our contributions include:

1) Obtaining raw CPU cache latency measurements of various systems
2) Performance characterization of measured latency data

IV. Methodology

For the cache hierarchy characterization, we will use existing tools that provide us with raw data. We initially looked for tools that utilize performance counters available in modern processors. These performance counters are part of the processors Performance Monitoring Unit (PMU) and this data can be obtained using widely available tools (e.g. Linux kernel profiling tool `perf`, `valgrind`, and Intel VTune). After researching various tools, we concluded that tools that use performance counters were mainly targeted towards profiling of application execution and therefore measure dynamic performance. Cache latency measurements are static performance metrics and we decided to shift our target towards those types of tools. We found several software projects that were suitable for cache latency measurements. The tools we selected are Intel MLC (Memory Latency Checker) [13], Tinymembench [14], and LMbench [15].

Intel MLC is a tool developed for measuring memory latency and bandwidth and has specific features to analyze latencies in multi-socket systems where Non-Uniform Memory Access (NUMA) is enabled. Measurements can be taken on an idling system or a loaded system. Intel MLC is closed source software but freely available for Linux and Windows.

Tinymembench is open source software and can measure memory latency and bandwidth. The software, which has not been updated for the last three years, has a small code base and can be compiled for various target platforms including x86, MIPS32, and ARMv7/8. The small and open-source code base allowed us to further inspect the code and apply patches where needed. We compiled Tinymembench on every target platform using GCC 7.5.0 and using `-O2`, `-march=native`, and `-mtune=native`.

LMbench3 is a suite of benchmark tools that dates from 1998 and has not seen an update since 2012. The large codebase is outdated and did not compile without modifications. The suite of benchmarks includes 49 tools of which one was our tool of interest. Instead of running the whole suite, we just used the `cache` tool of LMbench. We modified the `cache` tool to output raw data such that we can perform analysis later. LMbench was compiled with the same compiler infrastructure and flags as Tinymembench.

Name	CPU (cores/threads)	Cache size
Machine 1	Intel Core i7-6600U (2/2)	4 MB L3
Machine 2 (vm)	Intel Xeon Gold 6140 (1/2)	24.75 MB L3
Machine 3	Intel Core i7-8550U (4/8)	8 MB L3

TABLE I
MACHINES BENCHMARKED FOR THIS STUDY.

The fact that two of our tools were open source allows us to get a better understanding on how cache latency is measured for each benchmark. Both benchmarks take a similar approach where a buffer of memory is allocated and pseudo-random reads are performed on the buffer. System times can only be obtained with microsecond accuracy and tools to measure the execution time of many reads to calculate the latency of a single read with nanosecond precision. For example, Tinymembench performs 10 million reads for every sample. The latency of different cache levels can be measured by varying the buffer size. If the buffer size is larger than a cache level, cache misses on that level will occur which will influence the timing measurements. The Intel MLC software is not open source but according to the manual it uses a similar technique to extract latency data.

Our target is to obtain raw latency measurements (i.e. block size vs. read latency) using Intel MLC, Tinymembench, and LMbench3 on multiple machines. We will analyze all the data with our own software and compare the results of different tools and machines. LMBench and Tinymembench both scan over a range of buffer sizes while Intel MLC only tests for a single buffer size. We developed scripts to make all our benchmarks scan over a range of buffer sizes and run all desired measurements in an automated way.

V. Results

In this section, we will present the results we obtained using the methodology and benchmarks presented in Section II. We took results from three different machines which are listed in Table I. Cache sizes mentioned in Table I are provided by Intel [16]. Note that we had direct access to machine 1 and 3 while machine 2 is a virtual machine running on a shared server. Machine 1 and 2 are both running Linux while machine 2 runs on Windows.

A. Latency measurements

For every combination of machine and benchmark, we took 10 or more measurements. Each measurement contains a set of data points where each data point is a tuple with block size and an observed latency. It is important to note that Tinymembench reports latencies minus the L1D cache latency without reporting the L1D cache latency. Hence, those latency results are lower by some constant value. We would also like to mention that block sizes of Intel MLC are aligned to KB while the block sizes of other benchmarks are aligned to KiB. The mean and standard deviation of the latency per block size for machine 1/2/3 can be found in Figure 1/2/3.

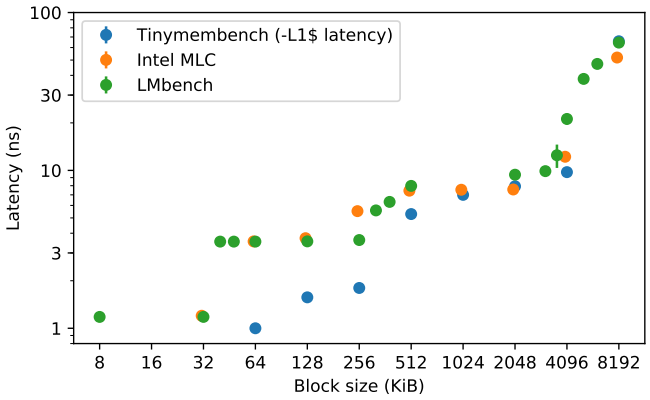

Fig. 1. Measured latencies for machine 1.

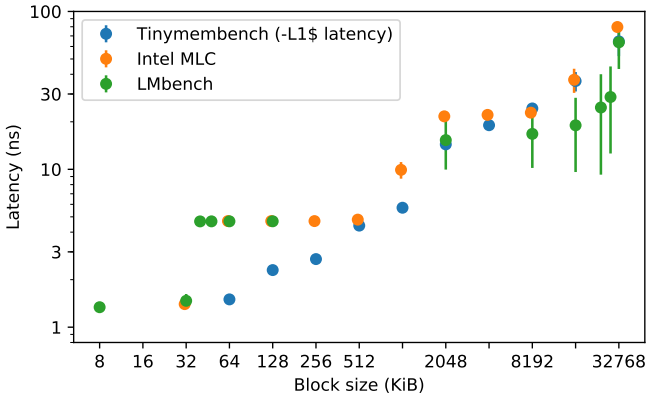

Fig. 2. Measured latencies for machine 2.

The results of Intel MLC and LMbench for machine 1, shown in Figure 1, clearly align and have a low standard deviation (mostly not visible in the graph). While Intel MLC scans over the block sizes exponentially, LMbench automatically takes extra data points around block sizes that are potential cache boundaries. A clear cache boundary can be observed at 32 KiB, which is the expected L1D cache size. Between 128 and 512 KiB, we also observe an increase in latency, but the transition is not as aggressive as the boundary at 32 KiB. We expect that the cache latency difference between L2 and L3 is not big and therefore the corresponding boundary is harder to measure. Finally, we do observe a transition between 4096 and 8192 KiB. Based on the specifications as presented in Table I, we expect a cache boundary at 4096 KiB. The data shows a slight increase in latency between 2048 and 4096 KiB before it makes a big step after 4096 KiB. We suspect that background processes pollute a small part of the shared L3 cache and therefore our benchmark sometimes suffers from L3 cache misses when the block size exactly matches the whole L3 cache size.

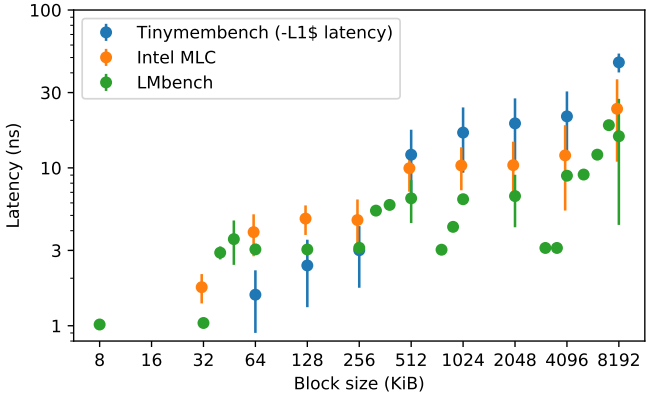

Fig. 3. Measured latencies for machine 3.

As mentioned earlier, the Tinymembench results for machine 1 are lower by a constant value, but the constant offset (which is supposed to be the L1D cache latency) is not reported. For block sizes of 32 KiB and less, Tinymembench reports a cache latency of 0, which is expected. Tinymembench starts reporting cache values from 64 KiB onwards, and significant latency steps can be observed between block sizes of 32-64 and 64-128 KiB. Therefore the expected L1D cache boundary is not as obvious as reported by Intel MLC and LMbench. In contrast, Tinymembench does show a clear L2 cache boundary at 256 KiB. All benchmarks agree with the provided specification that the L3 cache boundary is at 4096 KiB.

The benchmark results for machine 2, presented in Figure 2, show similar behavior as observed in the results for machine 1. Intel MLC and LMbench both strongly agree on an L1D cache boundary at 32 KiB and there are indications that point to an L2 cache boundary between 512 and 2048 KiB. LMbench data is missing for block sizes between 128 and 2048 KiB. We removed those data points because the chosen block sizes for different measurements would differ which resulted in single samples for a range of block sizes in that range. We consider the single sample data points not reliable and removed them with the gap in data as a result. Tinymembench also suggests a cache boundary at 32 KiB, but no clear conclusions can be drawn from the reported latencies between 64 and 1024 KiB. Tinymembench does show an L2 cache boundary at 1024 which aligns with the indications from the other benchmarks. Machine 2 has a larger L3 cache, as reported in Table I, and we extended our block size to 32 KiB. We do observe signs of a cache boundary between 16 and 32 MiB, which aligns with the specification. Our block sizes are powers of 2 and the reported cache size of 24.75 MB does not align. Therefore we are not able to pin down the cache boundary of the L3 exactly.

Machine 2 is a virtual machine on a shared server, and we were not sure if we could expect good measurement results. Modern virtualization techniques should reduce virtualization overhead to a minimum, and our results show that indeed we can perform cache latency measurements on a virtual machine.

Figure 3 shows the measurement data of machine 3. The results from machine 3 are significantly less stable and show higher deviation than the results from the two other machines. Intel MLC and LMbench indicate a cache boundary at 32 KiB,

but the two benchmarks do not agree as strongly as they did earlier. Both benchmarks also reveal a cache boundary at 256 KiB, but it is Tinymembench that provides convincing data for the L2 cache boundary. As indicated by the chip specification and suggested by Intel MLC and Tinymembench, the L3 cache boundary is at 8192 KiB.

We noticed a couple of interesting properties about the data of machine 3 that we would like to mention. Machine 3 is the only Windows machine and the data is much noisier than the measurements obtained on the other machines. It is unclear if the noise is caused by the operating system or by other factors. While Intel MLC yields fairly consistent results, the LMbench data is inconsistent when varying over the block size. Especially the LMbench latencies in the ranges 512-1024 KiB and 2048-4096 KiB are unlikely and significantly lower than latencies measured for smaller block sizes. On other machines, LMbench was able to yield good data and it is unclear why the data is inconsistent on machine 3. Finally, we would like to note that Tinymembench reports the highest latencies of all benchmarks for block sizes equal to or greater than 512 KiB. This observation is unusual since Tinymembench does not include the L1D cache latency and results are therefore lower than the other benchmarks. Again we are not able to point out a specific cause of this behavior and more measurements, especially on Windows machines, would be required to draw any conclusions.

B. Data analysis

Instead of manually inspecting the raw data, we developed analysis software that will automatically detect cache boundaries based on the measured data. As previously mentioned, each benchmark was run at least 10 times per machines 1, 2 and 3. To detect cache boundaries, we looked at edges within each measurement and identified patterns. These edges were then filtered both within each measurement and collectively per benchmark and machine. After completing edge filtering, the average latency across all measurements per benchmark per machine is taken and reported. The analysis steps used to select edges and cache boundaries are explained below.

1) Per-measurement filtering: First, edges between neighboring data points within a measurement are calculated by subtracting the 'current' latency from the 'next' latency and dividing by the 'current' latency. This value is referred to as edge size and is stored alongside the block size used by the benchmark to measure latency. If the 'current' latency value is greater than the 'next' latency value, the edges is set to zero because we know an edge is not occurring here. Similarly, if the current latency is less than one, this value is likely to be noise. In these cases, the 'current' latency value is set to one, and the 'next' latency value is set to its value plus one, thereby calculating an edge size equal to the first latency measured. After all of the edge sizes are determined, the average value is calculated. This average edge size multiplied by a scalar is used as an edge detection threshold in the next step. Table II shows he scalar used per benchmark, the average edges across all measurements and the standard deviation of the thresholds between measurements per machine.

Benchmark	Scalar	M1(μ, σ) (ns)	M2(μ, σ) (ns)	M3(μ, σ) (ns)
Intel MLC	0.75	0.681	0.654	0.535
		0.022	0.009	0.082
LMbench	1.00	0.326	0.625	–
		0.082	0.016	–
Tinymembench	1.25	0.673	0.629	0.804
		0.004	0.011	0.033

TABLE II
EDGE DETECTION THRESHOLDS PARAMETERS USED FOR THIS STUDY.

To calculate the edge detection threshold, the the scalars shown in Table II were multiplied by the average edge size for each measurement. Any rows with edges less than the edge detection threshold were dropped. Once these rows were dropped, this marks the end of data filtering conducted on a per-measurement basis.

2) Per-benchmark filtering: With the new data set, each of the unique edges was added to a candidate list. Clusters were then determined based off this list. A cluster was built if the blocks associated with the edge size were consecutive. These clusters represent possible candidates for the cache size. We used this method because we know that two caches will not be relatively close in size and two consecutive edges are likely to represent a single cache boundary. The primary factor used to determine the edge detection scalars was inspection of the size of a clusters. Specifically for the L1D and L2 caches, if a cluster was over three values, we increased the scalar in order to drop a value from the cluster, thereby creating two separate clusters. From here, the values were further filtered until only one block size remained in each cluster.

To filter the clusters, the next step was to count how many times each candidate edge was over the edge detection threshold across all measurements. If a cluster only had one value, we added this block size to the final list of detected edges. If there was more than one value within a cluster, we compared the counts of each edge. Whichever edge appeared more times after filtering based on edge size, this edge was kept, and the other edge was dropped. If two edges appeared the same number of times post edge size-based filtering, we compared the edge sizes and dropped whichever had a lower value. This process was repeated until only one candidate edge remained in each cluster.

These final edge candidates represent the cache levels detected by each benchmark for each machine. Only after determining these cache sized was the latency associated with the benchmark block size averaged. Tables III displays the experimental cache sizes and latencies for machine 1, while IV and V displays the cache sizes and latencies for machine 2 and machine 3, respectively. Due to the inconsistency of the data collected by LMbench for machine 3, this benchmark has been excluded in Table V

Comparing these results to the expected cache sizes based

Cache	Intel-MLC	LMBench	Tinymembench
L1D Size	32 KB	32 KiB	32 KiB
L1D Latency	1.20 ns	1.18 ns	0.00 ns
L2 Size	128 KB	256 KiB	256 KiB
L2 Latency	3.72 ns	3.62 ns	1.80 ns
L3 Size	4096 KB	4096 KiB	4096 KiB
L3 Latency	12.20 ns	21.18 ns	9.75 ns

TABLE III
AVERAGE CACHE ACCESS TIMES FOR MACHINE 1.

Cache	Intel-MLC	LMbench	Tinymembench
L1D Size	32 KB	32 KiB	32 KiB
L1D Latency	1.40 ns	1.47 ns	0.00 ns
L2 Size	1024 KB	128 KiB	1024 KiB
L2 Latency	9.92 ns	4.69 ns	5.71 ns
L3 Size	16384 KB	28672 KiB	16384 KiB
L3 Latency	36.93 ns	28.71 ns	36.23 ns

TABLE IV
AVERAGE CACHE ACCESS TIMES FOR MACHINE 2.

on visual inspection of latency measurements, we find the analysis tool aligns well with our expectations. As shown in Table III, for machine 1 the L1D cache is found to be 32 KB or KiB by all three tools. The L2 cache is found to be 128 KB by the Intel MLC benchmark and 256 KiB for both the LMbench and Tinymembench. While these cache sizes all fall within range of expected cache sizes, this discrepancy could likely be resolved with additional data points. During data analysis, both 128 and 256 KB were included in the cluster for L2 cache for the machine 1 Intel MLC benchmark, and collecting more measurements could would yield more counts for the candidate edges. This could affect which size cache is selected for machine 1's L2 cache. Including more block sizes around this L2 range could also help determine the cache boundary. Finally, all three benchmarks agree on an L3 cache size of 4096 KB or KiB, although this level is where we see the largest variance in latency. As previously mentioned, this could be due to other background processes polluting a part of the L3 cache.

Looking at Table IV, we again see that all three tools agree on an L1D cache size of 32 KB or KiB. However, the L2 results are not as neat. The analysis tool determined the L2 boundary for the LMbench data at 128 KiB, while Intel-MLC and Tinymembench data revealed edges at 1024 KB or KiB, respectively. This disagreement can likely be attributed to poor data quality collected by the LMbench tool within this range. For different measurements, the LMbench tool used different block sizes. To standardize the data, only block sizes used across all measurements were preserved. This resulted in an artificial edge being detected where no latencies are reported for block sizes between 128 and 2048 KiB as shown in Figure 2. For L3, the analysis tool reported a cache size of 16384 KB or KiB for Intel MLC and Tinymembench data, respectively, and a cache size of 28672 KiB for the LMbench data. Both of these values fall within the range we would expect based on visual inspection of the data, and again, it is not surprising the LMbench value differs due to the sporadicity of the data collected.

Cache	Intel MLC	Tinymembench
L1D Size	32 KB	32 KiB
L1D Latency	1.75 ns	1.18 ns
L2 Size	256 KB	256 KiB
L2 Latency	4.67 ns	3.01 ns
L3 Size	4096 KB	8192 KiB
L3 Latency	12.00 ns	46.60 ns

TABLE V
AVERAGE CACHE ACCESS TIMES FOR MACHINE 3.

For machine 3, only analysis of the Intel MLC and Tinymembench data were conducted also due to the inconsistent block sizes used by LMbench between measurements. Because there were such expansive and arbitrary gaps in the LMbench data for machine 3, too many rows needed to be removed to provide significant results. For the L1D cache, analysis of the Intel MLC and Tinymembench data agrees upon a cache size of 32 KB or KiB and an L2 cache size of 256 KB or KiB. These values match expected cache size based on inspection of the data in Figure 3. L3 cache is determined by analysis of Intel MLC data to be 4096 KB while Tinymembench data analysis yields an L3 cache size of 8192 KiB. This discrepancy is slightly surprising based on visual inspection of the results in Figure 3 because the Intel MLC and Tinymembench data appear to follow the same trends. For both benchmarks, 4096 and 8192 KiB appear in the L3 cluster during analysis. As discussed in relation to machine 1's results, additional data points could help resolve this discrepancy, either from more measurements collected, or from additional block sizes surrounding this range.

VI. FUTURE WORK

The scope of our project was limited and in this section, we would like to list some future work. First, we would like to discuss the measured data. This work could be extended by taking more measurements both per machine as well on more different machines. Per machine, we could make the step size for blocks smaller or add extra measurements around suspected cache boundaries as LMbench does. Taking more measurements might help us find the L2 cache boundary more accurately. For machine 2 our fixed block size step did not align with the expected L3 cache boundary. It would be interesting to use a fine grain block size step for machine 2 to measure the L3 cache boundary more accurately. Though since L3 is shared, we might not be able to get clean measurement results. We could also take more measurements on Windows machines to understand better what cache latency measurement data we can expect on a Windows operating system.

Our current analysis software processes data from different benchmarks for the same machine separately. As we have seen in Section V, not all benchmarks are equivalently good at

detecting certain cache boundaries. It would be interesting to explore the possibilities of a single analysis that takes the data of all benchmarks for one machine into account. Potentially the analysis could become more accurate, especially if we would add information about the strong and weak points of each benchmark.

Another analysis feature that we have been looking at was labeled "optimistic latency result". When benchmarks test with a block size that exactly matches a cache boundary, often a small increase in latency is measured. Probably background processes are polluting the same cache level and as a result, a small part of the reads is forwarded to the next memory level increasing the average read latency. Often the measurement with a slightly smaller block size gives a better impression of the cache latency of that specific level. The "optimistic latency result" would apply this knowledge to the analysis to return more optimistic latencies for every cache level.

Initially our plan was to perform a theoretical analysis on how our latency measurement results could potentially characterize or influence the performance of future in-cache computational accelerators as presented in [8], [9]. Due to the limited time span of this project we were not able to perform such an analysis in enough detail to include it in this report. This will remain as future work.

In an early stage of the project, we mentioned the possibility to perform an analysis of theoretical speedup obtainable by using PIM technologies (i.e. Amdahl's law [17]). For such work, we would assemble a set of benchmarks and select a set of rudimentary PIM instructions of our interest (e.g. `in_mem_copy`, `in_mem_zero`). Based on the profiles of the benchmarks and the chosen instructions of interest, we could provide quantitative results on ideal speedups using PIM. Unfortunately, we were not able to fit this part of the project in the time budget. Hence, this will remain future work for this moment.

VII. Conclusion

We have shown that we are able to measure cache latencies and sizes of various systems including virtual machines, bare metal, and machines with different operating systems. The three benchmarks we used (Intel MLC, LMbench, and Tinymembench) in principal work with the same concept, but output results would not always agree. L1D cache size could be found with high confidence on all machines with all benchmarks. While measuring the cache latency of L2 was not a problem, determining the cache size was more difficult. As mentioned earlier, we expect this is caused by the limited difference in cache latency between L2 and L3 cache. Intel MLC and LMbench are confident in revealing the L1D cache size and less precise in determining L2 cache size, this in contrast to Tinymembench which tends to point out the L2 cache boundary more clearly. The analysis tool developed to detect these cache boundaries firsts filters the data and then systematically removes the least likely candidates within clusters. Comparison of the detected cache sizes to the expected results as discussed in section V show that the analysis tool works well for both data generated across different benchmarks and on different machines. In general, analysis of results could be improved with increased robustness of the data set via more block sizes and additional measurements. Because of the dynamic edge filtering implemented, this tool could be applied to any cache latency data with minimal modification.

References

[1] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 1–14, 2018.

[2] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 541–552.

[3] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[4] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 1–13.

[5] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 383–396.

[6] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 380–392, 2016.

[7] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[8] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 481–492.

[9] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 397–410.

[10] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 336–348.

[11] M. Kang, E. P. Kim, M.-s. Keel, and N. R. Shanbhag, "Energy-efficient and high throughput sparse distributed memory architecture," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2015, pp. 2505–2508.

[12] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, 2016.

[13] "Intel mlc (memory latency checker) v3.8," https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[14] "Tinymembench, simple benchmark for memory throughput and latency," https://github.com/ssvb/tinymembench.

[15] "Lmbench - tools for performance analysis," http://www.bitmover.com/lmbench/.

[16] "Intel product specifications - intel ark," https://ark.intel.com/.

[17] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.