Understanding the Impact of Memory Access Patterns in Intel Processors

Mohammad Alaul Haque Monil*, Seyong Lee †, Jeffrey S. Vetter† and Allen D. Malony*
{mmonil, malony}@cs.uoregon.edu, lees2@ornl.gov, vetter@computer.org
* University of Oregon, † Oak Ridge National Laboratory, USA

***Abstract*—Due to increasing complexity in the memory hierarchy, predicting the performance of a given application in a given processor is becoming more difficult. The problem is worsened by the fact that the hardware needed to deal with more complex memory traffic also affects energy consumption. Moreover, in a heterogeneous system with shared main memory, the memory traffic between the last level cache (LLC) and the memory creates contention between other processors and accelerator devices. For these reasons, it is important to investigate and understand the impact of different memory access patterns on the memory system. This study investigates the interplay between Intel processors' memory hierarchy and different memory access patterns in applications. The authors explore streaming and strided memory access patterns with the objective of predicting LLC-DRAM traffic for a given application in given Intel architectures. Moreover, the impact of prefetching is also investigated in this study. Experiments with different Intel micro-architectures uncover mechanisms to predict LLC-DRAM traffic that can yield up to 99% accuracy for streaming access patterns and up to 95% accuracy for strided access patterns.**

***Index Terms*—Intel; Broadwell; Sky Lake; Cascade Lake; Memory Access Patterns; Memory Traffic Prediction**

I. Introduction

The disparity between the speed of CPU and memory continues to exacerbate the "memory wall" problem [15], especially in the context of modern multi/many-core CPUs and heterogeneous systems where shared memory is being used to link cores, processors, and accelerator devices together. While various innovations in memory hierarchy are helping to reduce the gap, multi-level cache hierarchy has been a core technique adopted by all processor manufacturers to battle memory wall issues. Clearly, aggressive cache hierarchy can be effective, but it also makes the interactions between the processor and the DRAM more complex to track, understand, and model.

Cache hierarchy design in micro-architectures has evolved significantly from just one level of fast cache to up to four levels of cache, with different sizes, line widths, associativity degrees, and sharing policies. Cache operation is complex and varies across processor manufacturers and within micro-architectures of the same manufacturer. It is also integrated necessarily with the memory management architecture supporting shared memory between other processors and devices, including cache coherency and virtual memory. Understanding the behavior of the memory system is a challenge. To do so requires knowledge of the operational policies, which are not entirely open to the community, as well as a means to observe effects of cache/memory actions, which is only partially visible through interfaces provided, such as hardware counters [16].

In the last decade, Intel has introduced several micro-architectures, from Westmere in 2010 (32nm) to the recently announced Tiger lake processors in 2020 (10nm). The micro-architecture innovations during this period include changes in the cache hierarchy, such as the size of the cache in different levels and it's policy. In recent Ice Lake processors, page table structure was also changed to use 5-level paging, while 4-level paging was used in the earlier processors. The number of memory controllers and memory channels were also changed in different micro-architectures. Clearly, all of these changes impact both the performance and energy consumption of the system and the applications that run on it. When trying to optimize these factors for an application, the real question is how to model them based on what we know of the micro-architecture. Even if we concentrate on just a part of the memory hierarchy, such as the memory traffic between LLC and DRAM, the problem is difficult and further complicated by the memory traffic patterns generated by an application.

Understanding the variation in different micro-architectures is important to evaluate LLC-DRAM traffic, which in turn is necessary for performance and energy prediction [8], [12], [17], [21]. Because memory accesses served by the cache are much faster than those by DRAM, there are clear performance and energy consumption implications of such a difference in memory access. Not only the hardware but also cache policy and prefetching policy play significant roles in deciding whether a memory request from a CPU core will be served by the cache or by the DRAM. Moreover, performance model, such as Roofline model is also dependent on LLC-DRAM traffic [11], [13], [22] and predicting this traffic can indicate compute memory intensity of a kernel.

The interplay between application and memory hierarchy is also an important factor to be considered for understanding memory traffic. Intel processors have a different mechanism for the read and write instructions. Moreover, the memory access pattern also plays a crucial role in determining the number of memory accesses. For example, the streaming access pattern would yield the best utilization of the cache hierarchy since most of the accesses will be served by the cache. The prefetching will also help this case. In contrast, this may not be true for the strided access pattern, depending on the stride length, cache, and prefetching.

This study investigates these factors via a systematic experimental evaluation of different memory access patterns with varying data sizes. The objective is to devise a robust prediction methodology that considers the hardware resources, cache policy for the read and the write instructions, prefetching mechanism, and application characteristics while investigating memory traffic generation from LLC towards DRAM.

This study reports the following contributions:

- Strategy to measure LLC-DRAM traffic in Intel processors using PAPI uncore counters and TAU.

- Prediction mechanism of memory traffic for an application with streaming access pattern.
- An analysis of strided access pattern of various stride length and prediction mechanism for this pattern.
- Investigate the impact of prefetching on both types of access patterns.
- Verify and measure prediction accuracy in different Intel micro-architectures.

II. Motivation

The complications in understanding the effects of the memory system on performance, in particular the cache hierarchy and its interactions with memory, can be demonstrated through a set of cases. The first case explored shows the difficulties in measuring traffic from LLC to DRAM. The second case highlights the importance of understating the differences of the read and write instructions with the cache hierarchy. Finally, the third case focuses on the difference between different micro-architectures.

```c
int stride = 1;
void vecMul(float *a, float *b, float *c, int n){
    for(int i = 0; i < n; i += stride)
        c[i] = a[i] * b[i];
}
int main( int argc, char* argv[] ){
    int n = 100000000;
    float *h_a, *h_b, *h_c;
    size_t total_size = n*sizeof(float);
    // Allocate and align the vectors
    h_a, h_b, h_c = align_allocate(total_size);
    // Initialize vectors
    for( int i = 0; i < n; i++ ) {
        h_a[i] = sin(i); h_b[i] = cos(i);
    }
    fill_cache();
    Start_memory_counters();
    vecMul(h_a, h_b, h_c, n);
    Stop_memory_counters();
    report_counter();
    free(h_a); free(h_b); free(h_c);
}
```

Listing 1: Vector multiplication

A. Observing LLC and DRAM traffic

Consider the case of vector multiplication shown as a C code skeleton in Listing 1. There are three arrays denoted by h_a, h_b, and h_c. These arrays are allocated 100M floating-point elements and aligned using the posix_memalign() function. The vectors h_a and h_b are then initialized with some values. The fill_cache() (detail is not shown) function fills the cache with some other data so that the vector multiplication is not interfered with by any previous data from the initialization phase that resided in the cache. The function vecMul() is then called to multiply h_a and h_b elements and store them.

To count bytes transferred between LLC and DRAM, PAPI [20] preset counters are used in the Start_memory_counters(), Stop_memory_counters(), and report_counter() functions. For instance, the PAPI_L3_TCM event counter refers to the total cache miss at the last level of cache which includes load and store misses of both data and instructions. Since the given code is a sequential streaming access pattern (i.e., stride = 1) and is aligned, it is expected to be very cache-friendly. Therefore, a straightforward way to calculate the traffic is by multiplying the cache miss event at LLC by the cache line length (e.g., 64 bytes).

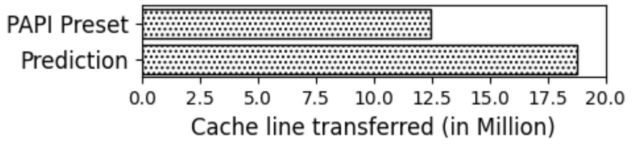


Fig. 1: Comparison of prediction vs PAPI preset event.

The expected data transfer for 100M data size is 2*100M + 1*100M = 300M floating points since there are two loads and one store. This results in the total byte transfer being 300M * size_of_float = 1200M bytes. When divided by the cache line length for the Broadwell micro-architecture, a total of 18.75M cache lines is expected to be transferred between LLC and DRAM. The comparison between predicted and measured by PAPI is shown in Fig. 1. Oddly, the PAPI event count for last level cache miss is found to be a total of 12.45M, approximately two-thirds of the prediction. This result is the same if off-core events are used from papi_avail tool. The reason behind this discrepancy is that PAPI counters in Broadwell do not show the cache misses that occur with the store instruction. This is confirmed by modifying the vecMul kernel to remove the store instruction and keeping two load instructions only. In this case the results matched.

Interestingly, the earlier Sandy Bridge micro-architecture does include the write back in the off-core events, but the recent Ice Lake again does not count the write back (see ”Off-core Response Performance Monitoring” in Chapter 18 of Volume 3 of the Intel Architecture SW Developer’s Manual [3]. Simply put, this case demonstrates that designing a prediction strategy for evaluating memory traffic must proceed cautiously, taking into account not only the memory access pattern, but also whether and how the micro-architecture makes visible events to calculate outcomes.

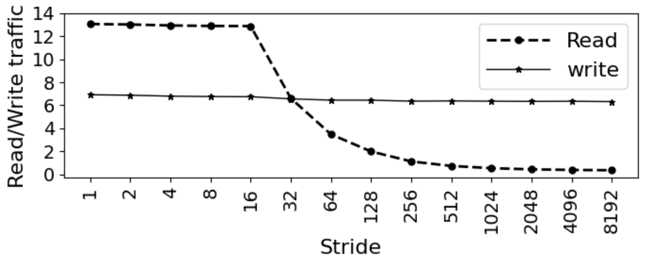


Fig. 2: Read and write traffic (in Million) with varying stride.

B. Observation for strided execution

Things become more complicated with non-sequential memory access. A common case is strided execution as exemplified by Listing 1. Fig. 2 shows results from the method presented in §III-A, where read and write counts are measured from the uncore elements. Again, a Broadwell micro-architecture is used. For stride=1, it provides a 2x read count when compared to write. After stride 16 the read count drops by half, which is expected since the cache line length is 64 bytes (16*size_of_float = 64bytes). However, it is interesting to notice that the write count is unchanged even though the number of access is reduced significantly with higher stride. Why? Does this occur with other Intel micro-architectures? An investigation is necessary to understand such behavior and create prediction models that take the full context (e.g., traffic pattern, counter availability, cache line width) into account.

C. Difference between micro-architecture

Certainly, differences between micro-architectures can impact prediction models and must be taken into account. Uncore components can change thereby influencing the measuring technique and ability to accurately measure traffic between the LLC and DRAM. Moreover, prefetching mechanism and cache policy can also be changed. Consider another twist in our prediction methodology. To what extent can conclusions drawn from one micro-architecture be used to reason about the LLC-DRAM traffic in another micro-architecture? If possible, it is desirable to design prediction mechanisms that work across different micro-architectures.

The cases above guide the investigation presented in this paper. The next section describes our prediction methodology and the criteria we use to evaluate success. The outcomes from experiments on different Intel micro-architectures are used to evaluate advantages and disadvantages of our approach.

III. Methodology

Our goal is to create a robust model for predicting the performance of LLC-DRAM traffic that can provide insight into the interplay between memory access patterns and the cache hierarchy in Intel processors. To do so, we developed a LLC-DRAM traffic measurement mechanism that can take into account the differences in micro-architecture support for performance counters. That mechanism is used to analyze traffic performance for different access patterns: streaming and strided. The impact of prefetching is also discussed.

A. Measuring LLC-DRAM traffic using PAPI and TAU

LLC-DRAM traffic measurement requires observation of hardware actions associated with the cache-memory interface. Since PAPI preset counters and off-core counters from papi_avail tool are insufficient for measuring both read and write traffic at this interface, it is necessary to use events for the uncore component. Uncore events can be counted from different uncore components [6]. It can be counted from the caching agent (bdx_unc_cboXX) which exists one per L3 cache slice. (For instance, a Xeon E5-2683 v4 processor of the Broadwell family has sixteen CBo.) Measurement can be also done through a home agent for each memory controller (bdx_unc_haX). However, in this study, events for the integrated memory controller (bdx_unc_imcX) are used. In the case of the Xeon E5-2683 v4 processor used in our study, there are two memory controllers and two memory channels for each controller. So in total, there are four components. The event name format is: bdx_unc_imc[0 or 1 or 4 or 5]:: UNC_M_CAS_COUNT: [RD or WR]:cpu=x. Here, x stands for the physical CPU core where the code is running. The detail of the read/write events of CAS_COUNT is found in "Table 2-120. Unit Masks for CAS_COUNT" of Intel document for Xeon E5 v4 [4].

Because different micro-architectures will have different settings, these events need to be enabled and are not activated by default. PAPI paranoid needs to be set ("echo 0 > /proc/sys/kernel/perf_event_paranoid" as root). After setting the PAPI paranoid value, the papi_native_avail tool shows the uncore events. To evaluate the measured data, it is necessary to compare it to a known value. In this study, events only for vecMul() function of Listing 1 need to be counted. This can be done by manually instrumenting PAPI native counters. However, TAU [19] is used in this research since it can generate function-wise profile and counter data. TAU needs to be configured with PAPI and used with native counter support. Using these tools, a script is prepared which can generate the traffic measurement for the required function.

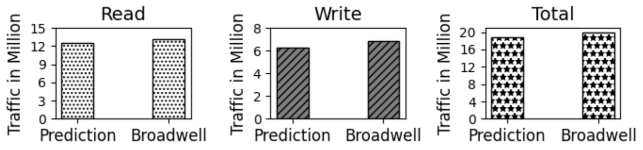


Fig. 3: Comparison of prediction vs PAPI uncore event.

B. Streaming access pattern

The streaming access pattern is one of the most common patterns in applications and yields the best cache performance. The prediction mechanism for streaming access pattern is based on the size of the array being accessed, the cache line width, and the number of read and write operations. For the vector multiplication case, Fig. 3 shows the cacheline transfer comparison between the prediction and the transfer measured through uncore counters. Unlike Fig. 1, the transfer is found to be close with 93.7% accurate for the total count, 91.5% for write, and 94.9% for read traffic. One observation from Fig. III-B is that for both read and write traffic the measured values are higher than the predicted one. We consider the accuracy to be reasonable and will continue to use the approach in the rest of the study.

C. Strided access pattern

In contrast to streaming, a stride access pattern is more complicated. Although still quite common in applications, memory traffic performance in stride scenarios is less well understood. Figure 2 shows the measured transfer of cache lines for store instruction. Interestingly, it did not change with varying stride. There are two major concerns with these results. First, the number of writes seems to be equal to the data structure size irrespective of stride, which means that the number of cache line transfers for store instruction does not depend on the number of actual data access. This seems dubious. To put things in perspective, consider the following example. If there is an array of size 100M and stride size is 4096, there are a total of 24414 array element accesses. For a stride of 1, the number of array element accesses is 100M. It seems highly unlikely that the results of 6.5M cache line transfer for store instructions would be the same for both cases. The second concern is to find out if there is a limit for this kind of behavior. Figure 4 answers the above concerns, but we need to better understand the read and write strategy of Intel.

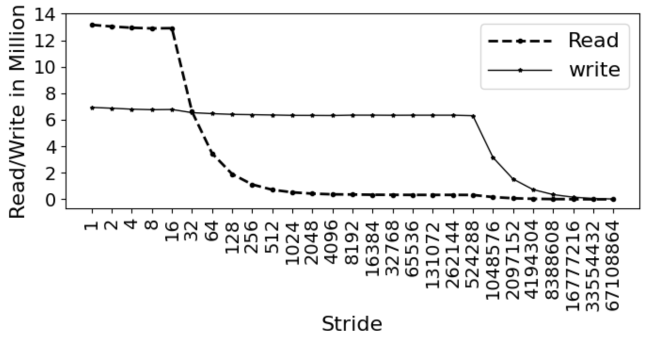


Fig. 4: Explanation of cache line transfer for read and write from DRAM with varying stride up to higher number.

1) Write strategy: Figure 4 provides clues about the write strategy. After increasing the stride up to 256MiB (67108864 * size_of_float), the change in cache line transfer becomes apparent versus what is seen in Fig. 2. Up to stride size of 2MiB (524288 * size_of_float), the number of cacheline transfer seems to be constant. After that point, stride doubling reduces the number of transfers by half. There are two reasons for this behavior. The first reason is initialization. In Listing 1, the vector h_c is not been initialized and for this reason, page zeroing occurred. Uninitialized arrays are initialized with the first write up a page. When initialization takes place, all lines in a page become dirty irrespective of the number of actual writes. For this reason, the number of cache line transfer has no relation to the number of data accesses. Page zeroing occurs to avoid information leakage from the previous content [2]. The default page size in a 64 bit OS (Centos-7 in our case) for Intel processors is 4KiB. However, the default page size did not have any effect. This is because Linux also supports huge page sizes for Intel processors, which are 2MiB, 4MiB, or 1GiB (4MiB is for 32 bit OS). Most versions of Linux support ”transparent huge pages” which enables the capability of using huge pages when possible. Because 100M arrays are used, 2MiB pages are selected. If an array smaller than 2MiB is used, the default page size of 4KiB is selected. If the array for store (Array h_c for Listing 1) is not initialized, then the minimum data to be written is equal to the page size in use.

2) Read strategy: Reading from DRAM is less expensive (compared to write) and is easier to interpret. Since prefetching is disabled in our experiments, cache line size becomes the deciding factor for the load instruction. In Fig. 4, the line for read traffic shows that the number of cache line transfers are consistent up to stride size of 16 (16 * size_of_float = 64Bytes) which is equivalent to the cache line size. This is because, every time a data is read from DRAM, the total cache line is read. However, for stride size larger than 16, the cache line transfer number reduces by half due to stride doubling. From stride 128 onwards, the ratio slips from two and becomes a straight line till the 2MiB cut-off line. To make an accurate prediction, instead of considering reduction by half, the ratio should be considered for those stride sizes.

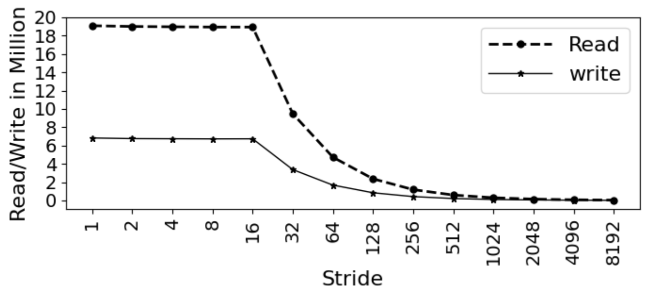


Fig. 5: Impact of initialization on cache line transfer for read and write. Both read and write traffic are impacted.

D. Strided access pattern with initialization

Since initialization of the array used for storing the result plays a major role in cache line write transfers, we can modify Listing 1 to initialize h_c with a value. Running the experiment again with varying stride size, the results in Fig. 5 are obtained. After initialization, the change of the cache line transfer for read and write is visible.

1) Write strategy: Since page zeroing is not a factor for this case, write strategy is determined by the available caching method. Intel has 6 types of caching methods possible (see ”11.3 Methods of Caching Available” in the Intel document [3]). For the *write combining* method, a write combining buffer of size 64 Bytes is used, the same as the cacheline. If write combining buffer is not used, the write strategy is dependent on cache line size. Thus, a prediction strategy with 64 Bytes caching covers both cases. The write line in Fig. 5 is similar to read behavior where doubling stride after stride 16, the number of cache line transfer reduces by half. A prediction strategy can be formed from this observation.

2) Read strategy: Figure 5 shows an interesting case for the cache line transfer for read. If Fig. 4 and Fig. 5 are compared, it is visible the read counts are much higher in the later. To be exact, the read counts are 1.5x higher with initialization than without. The extra cache line transfer counts are coming from the write strategy. When initialized, the Intel caching method loads a cache line for a cache miss on write. For this reason, the read counts are as if three arrays are being read. Apart from this distinction, the behavior is the same with a non-initialized version in which cache line is the deciding factor. Using this observation a prediction strategy can be established.

E. Impact of prefetch in Intel hardware

Prefetching attempts to increase the cache hit rate to improve performance [14]. Intel hardware has an aggressive prefetching mechanism and some of it is disclosed. The prefetching mechanism for Broadwell micro-architecture [1] uses four types of h/w prefetchers. On every processor core, there is a Model-Specific Register (MSR) that can control these prefetchers. Four bits (0-3) can be set/not-set to enable and disable these prefetchers. The first bit is responsible for fetching additional cache lines to L2 cache. The second bit enables to fetch adjacent cache line which makes the total cache line length double (128 bytes). The third and fourth bits are for fetching the next cache line to L1-D cache and for sequential load history-based fetching. In our work, a custom prefetch enabler/disabler is used [5]. The impact of prefetching is discussed for both initialized and non-initialized case below.

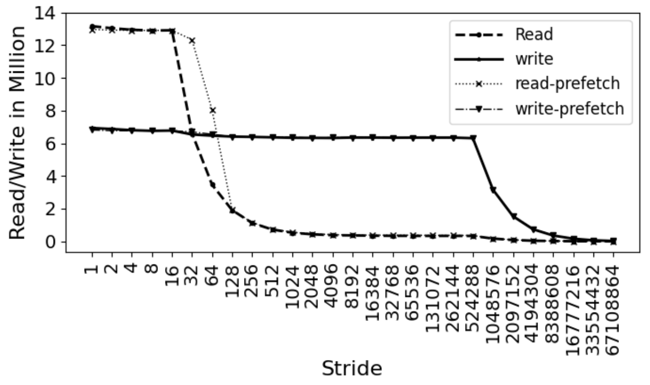


Fig. 6: Impact of large page size for different stride size with prefetching enabled. No impact on write strategy.

1) Impact of prefetching in non-initialized array: Prefetching needs to be understood to prepare a prediction mechanism that can provide reasonable accuracy for most, if not all, cases. Figure 6 presents the comparison between cache line transfer count for read and write with prefetching enabled and disabled. There is no visible impact for the write strategy since write does not depend on cache line for this case. However, there is a notable difference in read traffic when prefetch is enabled. There are two major differences observed. The first difference is that read traffic is not halved at stride 32, rather it stayed similar to streaming access. This is because adjacent cache lines are pulled in, giving the effect of a line twice as wide. This is equivalent to a stride of 32. The second observation is that for stride size 64, the traffic did not halve. This is because of the first prefetch criteria where along with two cache lines additional cache lines also are fetched. To provide a prediction, the number of data access at stride 64 should be multiplied by three for a reasonable prediction.

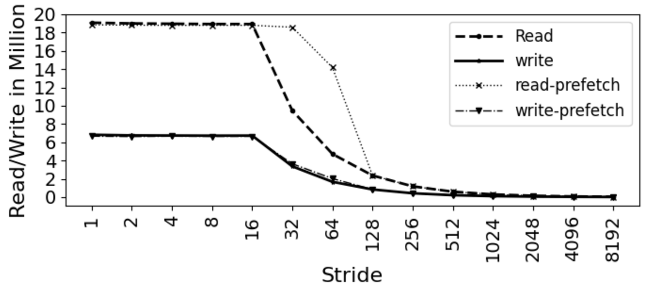


Fig. 7: Impact of prefetching on initialized arrays.

2) Impact of prefetching in initialized arrays: The impact of prefetching on initialized arrays is portrayed in Fig. 7. There is no significant change for write traffic. The visible difference in read traffic is due to the same reasons for the non-initialized case. Prediction can also be made in the same way.

F. Prediction criteria

A demonstration of our prediction is given in Table I for varying stride with prefetch enabled for an initialized array. Data is in million. Since the write array is initialized, the predicted read traffic is calculated for three arrays. Since prefetch is enabled, from stride 1 to 32, the predicted read traffic is kept same as streaming access. Change starts from stride 64. For stride 64, for read prediction, the actual data access is multiplied by three to include prefetching impact as suggested by the Broadwell prefetching mechanism. For stride 128, the prediction divides the streaming access data by 8 because the impact of prefetching vanishes at this point and a regular strided pattern is followed. After that, every data is halved for each doubling stride size. Predicted write traffic is kept same until stride 16 and then reduced by half. This prediction leads a good accuracy for this case.

TABLE I: Prediction for initialized array with prefetching.

Stride	Read	Predict R	Acc.	Write	Predict W	Acc.
1	18.862	18.750	99.4	6.665	6.250	93.8
2	18.814	18.750	99.7	6.625	6.250	94.3
4	18.800	18.750	99.7	6.663	6.250	93.8
8	18.804	18.750	99.7	6.638	6.250	94.2
16	18.803	18.750	99.7	6.645	6.250	94.1
32	18.591	18.750	99.1	3.645	3.125	85.7
64	14.190	14.063	99.1	2.030	1.563	77.0
128	2.365	2.344	99.1	0.840	0.781	93.0
256	1.184	1.172	99.0	0.421	0.391	92.7
512	0.594	0.586	98.6	0.213	0.195	91.9
1024	0.298	0.293	98.4	0.107	0.098	91.3
2048	0.149	0.146	98.1	0.059	0.049	83.4
4096	0.075	0.073	97.2	0.031	0.024	78.0
8192	0.038	0.037	96.7	0.016	0.012	76.7

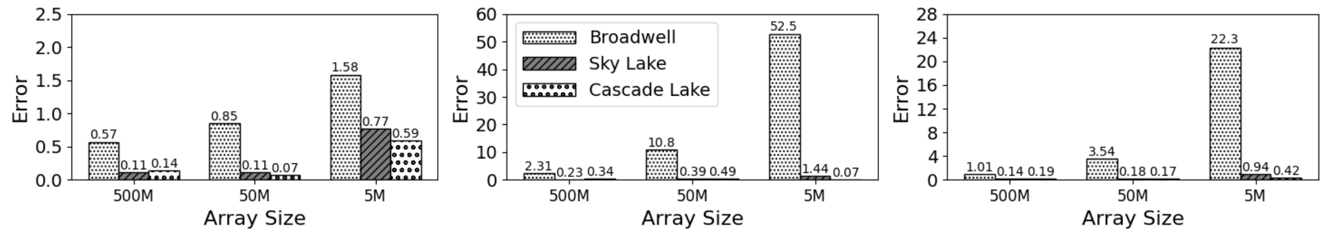
TABLE II: Prediction for non initialized and no prefetching.

Stride	Read	Predict R	Acc.	Write	Predict W	Acc.
1	6.565	6.250	95.2	3.641	3.125	85.8
2	6.531	6.250	95.7	3.613	3.125	86.5
4	6.471	6.250	96.6	3.477	3.125	89.9
8	6.478	6.250	96.5	3.527	3.125	88.6
16	6.449	6.250	96.9	3.530	3.125	88.5
32	3.307	3.125	94.5	3.348	3.125	93.3
64	1.741	1.645	94.5	3.249	3.125	96.2
128	0.952	0.914	96.0	3.214	3.125	97.2
256	0.563	0.537	95.5	3.211	3.125	97.3
512	0.363	0.358	98.7	3.146	3.125	99.3
1024	0.264	0.276	95.5	3.177	3.125	98.4
2048	0.216	0.230	93.6	3.137	3.125	99.6
4096	0.194	0.209	92.2	3.164	3.125	98.8
8192	0.178	0.209	82.7	3.167	3.125	98.7

Prediction data for non initialized array (size 50M) without prefetching is presented in Table II. Since the stride size is smaller than 2MiB, the predicted write traffic is considered the same. For predicted read traffic, the data is kept same as streaming access up to stride 16. For 32 stride the prediction halved. As mentioned earlier, from stride 64 onward the read traffic starts plateauing, and hence the ratio observed from Fig. 4 (size 100M) is used for predicting read traffic. For example, the ratio of read traffic for 128 and 256 stride is 1.7 instead of two. (Further explained in Appendix §VIII-C).

IV. Experimental Setup

The section discusses the micro-architectures of Intel which are used in this study. The machines oswald00, quad00, and apachepass are presented in Table III and are the part of ORNL Experimental Computing Laboratory (ExCL). For all these nodes, the default page size is 4KiB. The Broadwell machine



(a) Read. (b) Write. (c) Total.

Fig. 8: Prediction accuracy of traffic between LLC-DRAM for different micro-architectures.

has two memory controllers where each controls two memory channels. On the other hand, Sky Lake and Cascade Lake machines have three memory controllers that control a total of six memory channels. GCC 9.1 is used for experiments and also used to build TAU and PAPI. All the experimentation is done using Listing 1 with necessary modifications.

TABLE III: Micro-architectures.

Name	Year	Processor	LLC	Machine
Broadwell	2016	Xeon(R) E5-2683 v4	40MB	oswald00
Sky Lake	2017	Xeon(R) Silver 4114	14MB	quad00
Cascade Lake	2019	Xeon(R) Gold 6248	28MB	apachepass

V. Experimental Results

This section presents the verification of the conclusion drawn in the methodology section. To verify, predicted traffic for read and write is compared against measured traffic not only in different micro-architectures but also with different data sizes. In the methodology section, observations were made on an array size of 100M. Those observations are verified by using three array sizes which are 500M, 50M, and 5M. Each array size is tested against each micro-architecture. The experimental evaluation is done for five cases. At first, the streaming access pattern is verified for different micro-architectures. Then strided access pattern is verified in four ways, 1) non-initialized array where prefetching is disabled, 2) non-initialized array where prefetching is enabled, 3) initialized array with prefetching disabled, and 4) initialized array with prefetching enabled. In all the graphs in this section, data for Broadwell is presented by white dotted bars, Sky Lake is presented by grey slashed bars, and Cascade Lake is presented by white circled bars (given in Fig. 8b). The formula of error is, error = Absolute[(measured-predicted)/measured*100] and formula for accuracy is, accuracy = [100 - error].

A. Comparison of streaming access pattern

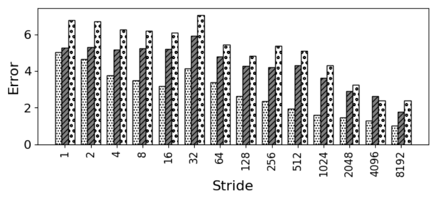
Figure 8 presents the comparison with predicted traffic for streaming access pattern for different micro-architectures. In this figure, predicted and measured read, write, and total traffic is compared using the error metric. Figure 8a shows, all three micro-architectures yield very high accuracy for read traffic prediction. Broadwell provided 99%, Sky Lake provided, 99.6% and Cascade Lake provided 99.7% average accuracy considering the accuracy for all data sizes. Write traffic prediction is presented in Fig. 8b where Sky Lake and Cascade Lake provided accuracy of 99.3% and 99.7% respectively. However, Broadwell provided 79.2% overall accuracy because of poor accuracy (48.5%) for 5M array size. This indicates the processor is generating extra write traffic towards DRAM for smaller array sizes. Based on the data, it is evident that this behavior improved in the later Sky Lake and Cascade Lake processors. Overall (read + write) traffic comparison is shown in Fig. 8c. As expected, the Sky Lake provided 99.5% and Cascade Lake provided 99.7% accuracy. Because of high inaccuracy for the smallest array size, Broadwell provided 91.1% accuracy overall. A total of 25 cases out of 27 provided accuracy which is more than 94%. For better representations, total traffic is presented in the later comparisons.

B. Strided access pattern with non-initialized array

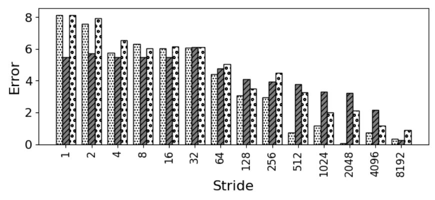
Prediction accuracy for non-initialized array (e.g., array h_c for Listing 1) is verified and presented in Fig. 10 for varying stride size. These experiments are done for three data sizes in three micro-architectures for each stride. Figure 10 also provides a comparison with prefetching enabled and disabled. The prediction mechanism for strided access with and without prefetching is discussed in §III-F. Overall accuracy is calculated by averaging all strides' accuracy.

1) Prefetching disabled: Figures 9a, 9b, and 9c present the comparison for prefetching disabled cases. In Fig. 9a, for 500M array size, Broadwell showed 97.2% accuracy, Sky Lake showed 95.7% accuracy, and Cascade Lake showed 95.7% accuracy. There is no case where accuracy is less than 92%. In Fig. 9b, for 50M array size, Broadwell showed 96.2% accuracy, Sky Lake showed 95.8% accuracy, and Cascade Lake showed 95.8% accuracy. In this figure, for all 42 cases, accuracy is higher than 91%. In Fig. 9c, for 5M array size, Broadwell showed 83.2% accuracy, Sky Lake showed 92.7% accuracy, and Cascade Lake showed 92.7% accuracy. The reason for lower accuracy in Broadwell is due to lower accuracy in predicting write traffic. Since the data for streaming access pattern continues up to the stride size equivalent to cacheline size (which stride 16 or 64bytes) and accuracy is low for streaming access pattern in Broadwell for smaller array size, accuracy stays low until stride 16. After stride 16, accuracy seems to improve for Broadwell. Like streaming access pattern, this category also yields high accuracy for Cascade Lake and Sky Lake while observing lower accuracy for the lowest array size for Broadwell.

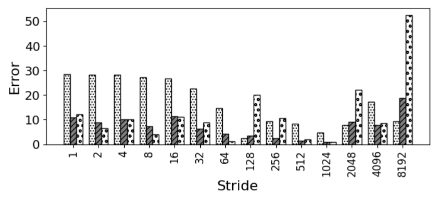
2) Prefetching enabled: Prediction mechanism with prefetching enabled is based on the disclosed prefetching document of Intel Broadwell micro-architecture [1]. Since then prefetching mechanism changed in later micro-architectures [2]. Figures 9d, 9e, and 9f present accuracy



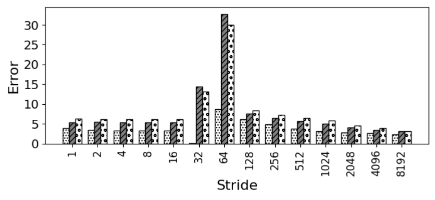
(a) 500M-prefetching disabled.



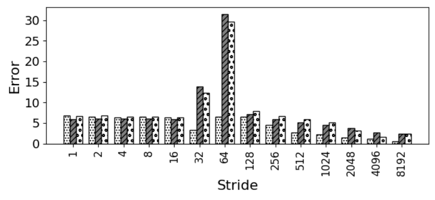
(b) 50M-prefetching disabled.



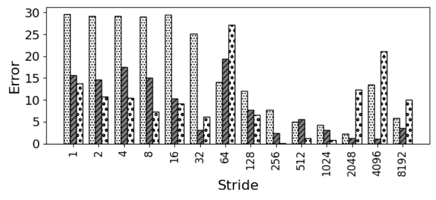
(c) 5M-prefetching disabled.



(d) 500M-prefetching enabled.



(e) 50M-prefetching enabled.



(f) 5M-prefetching enabled.

Fig. 9: Prediction accuracy for non-initialized arrays with prefetching enabled and disabled.

of prediction when prefetching is enabled. In Fig. 9d, for 500M array size, Broadwell showed 96.3% accuracy, Sky Lake showed 92.2% accuracy, and Cascade Lake showed 92.2% accuracy. In Fig. 9e, for 50M array size, Broadwell showed 95.6% accuracy, Sky Lake showed 92.4% accuracy, and Cascade Lake showed 92.4% accuracy. In Fig. 9f, for 5M array size, Broadwell showed 83.1% accuracy, Sky Lake showed 91.4% accuracy, and Cascade Lake showed 91.4% accuracy. Lower accuracy is observed for Broadwell for low array size which is explained earlier. Prefetching prediction based on Broadwell did not provide high accuracy for stride 32 and 64 for Sky Lake and Cascade Lake. The data suggests that the low accuracy for those stride sizes are introduced by the change in prefetching mechanism. However, for 500M and 50M array size Broadwell provided high accuracy for those stride sizes (at least 92%).

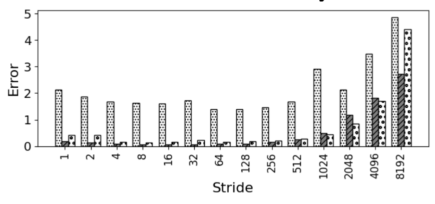
C. *Strided access pattern with initialized array*

In Fig. 9, prediction mechanism demonstrated in §III-F for initialized array (e.g., array h_c for Listing 1) is verified with prefetching enabled and disabled. When array is initialized, the constant write traffic case disappears.

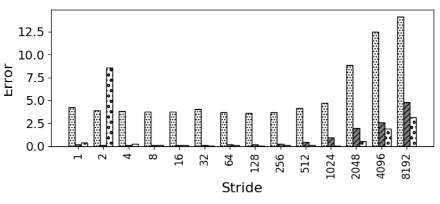
1) Prefetching disabled: Figures 10a, 10b, and 10c present the comparison for prefetching disabled cases. In Fig. 10a, for 500M array size, Broadwell showed 97.9% accuracy, Sky Lake showed 99.5% accuracy, and Cascade Lake showed 99.5% accuracy. In Fig. 10b, for 50M array size, Broadwell showed 94.4% accuracy, Sky Lake showed 99.1% accuracy, and Cascade Lake showed 99.1% accuracy. Even though some high inaccuracies are visible in Fig. 10b, the accuracy did not go below 87%. In Fig. 10c, for 5M array size, Broadwell showed 73.8% accuracy, Sky Lake showed 97.8% accuracy, and Cascade Lake showed 97.8% accuracy. Error is higher with a higher stride in some figures. This is because in a higher stride, the number of access is very low and small variation seems larger. Moreover, Broadwell yields low accuracy for the smallest array size. However, the error is constantly high which provides a chance to adjust the prediction criteria.

2) Prefetching enabled: Figures 10d, 10e, and 10f present the comparison for prefetching disabled cases. In Fig. 10d, for 500M array size, Broadwell showed 98.2% accuracy, Sky Lake showed 95.9% accuracy, and Cascade Lake showed 95.9% accuracy. In Fig. 10e, for 50M array size, Broadwell showed 94.1% accuracy, Sky Lake showed 95.4% accuracy, and Cascade Lake showed 95.4% accuracy. In Fig. 10f, for 5M array size, Broadwell showed 69.1% accuracy, Sky Lake showed 93.7% accuracy, and Cascade Lake showed 93.7% accuracy. Similar prefetching behavior like non-initialized cases for Cascade Lake and Sky Lake is observed for 32, 64, and 128 strides due to Broadwell specific prediction mechanism.

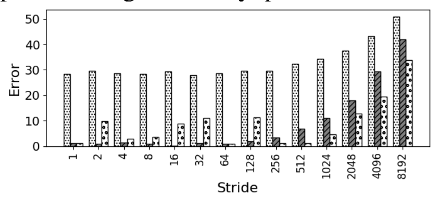
While most cases provided high accuracy, prediction mech-



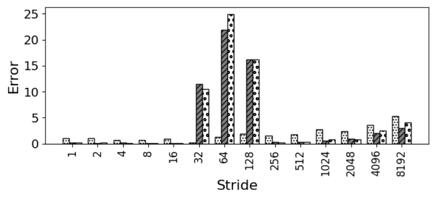
(a) 500M-prefetching disabled.



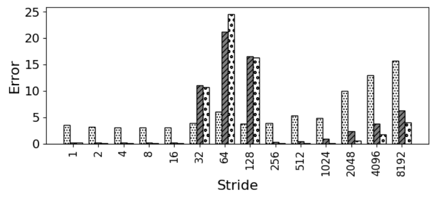
(b) 50M-prefetching disabled.



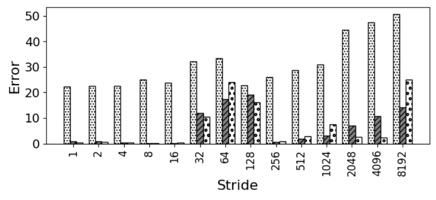
(c) 5M-prefetching disabled.



(d) 500M-prefetching enabled.



(e) 50M-prefetching enabled.



(f) 5M-prefetching enabled.

Fig. 10: Prediction accuracy for initialized arrays with prefetching enabled and disabled.

anism for small array size for Broadwell and prefetching for Sky Lake and Cascade Lake has room for improvement.

VI. Related Works

Two previous studies looked deeper into memory access pattern to come up with an analytical model [18], [23]. Peng et al. [18] introduced analytical model for different memory access pattern and interfaced those model in a cycle accurate simulation to find memory traffic. On the other hand, Yu et al. [23] investigated applications vulnerability by using an analytical model that uses cache hierarchy to predict memory traffic for different access patterns. However, both of these works are based on simulation and fall short when compared to complex cache hierarchy of modern processors. For this reason, investigating the change in different micro-architecture is important. Using Intel advisor tool, Marques et al. [13] analyzed performance of benchmark application to understand and improve cache performance. In [7], Alappat et al. investigated Intel Broadwell and Cascade Lake processors for understanding the cache behaviour. Hammond et al. investigated Intel Sky Lake processor [9]. Hofman et al. also investigated different micro-architectures [10], [11]. Understanding cache performance is important for Roofline model [22]. Predicting traffic between LLC and DRAM provides higher accuracy when predicting execution time and energy consumption. Monil et al. [17] studied memory contention in shared memory heterogeneous system where traffic between LLC and DRAM plays a crucial. Allen et al. [8] investigated impact of different memory access patterns on GPU. Lee et al. introduced automated modeling and prediction framework [12] where determining LLC-DRAM traffic will improve accuracy. The same goes for the study by Umar et al. [21].

VII. Conclusion and future work

The interplay between application's memory access pattern and cache hierarchy of Intel micro-architectures is investigated in this study. Through experimentation a prediction mechanism that yields reasonable accuracy is introduced from the observation from one micro-architecture which in turn is tested in two other micro-architectures. A prediction mechanism that is built from the observation from Broadwell processor yielded reasonable prediction accuracy when tested against Sky Lake and Cascade Lake. However, prediction mechanism for prefetching enabled cases in Sky Lake and Cascade Lake has room for improvement. The authors foresee formulating an analytical model for Intel processors that can predict traffic between LLC and DRAM. Moreover, predicting energy consumption and performance by taking traffic between LLC and DRAM into consideration is also in the authors' future plan.

References

[1] Disclosure of hardware prefetcher control on some intel processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html. Accessed: 2020-09-01.

[2] Explanation of using different page size. https://community.intel.com/t5/Software-Tuning-Performance/Explanation-of-LLC-to-DRAM-write-count-in-Haswell/m-p/1205752#M7638. Accessed: 2020-09-01.

[3] Intel® 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Accessed: 2020-09-01.

[4] Intel® xeon® processor e5 and e7 v4 product families uncore performance monitoring reference manual. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-e7-v4-uncore-performance-monitoring.html. Accessed: 2020-09-01.

[5] Tool to enable and disable prefetch in intel processor. https://github.com/deater/uarch-configure/tree/master/intel-prefetch. Accessed: 2020-09-01.

[6] Uncore performance monitoring of intel micro-architecture. https://software.intel.com/content/www/us/en/develop/blogs/documentation-for-uncore-performance-monitoring-units.html. Accessed: 2020-09-01.

[7] C. L. Alappat, J. Hofmann, G. Hager, H. Fehske, A. R. Bishop, and G. Wellein. Understanding hpc benchmark performance on intel broadwell and cascade lake processors. *arXiv preprint arXiv:2002.03344*, 2020.

[8] T. Allen and R. Ge. Characterizing power and performance of gpu memory access. In *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, pages 46–53. IEEE, 2016.

[9] S. Hammond, C. Vaughan, and C. Hughes. Evaluating the intel skylake xeon processor for hpc workloads. In *International Conference on High Performance Computing & Simulation (HPCS18)*, pages 342–349, 2018.

[10] J. Hofmann, D. Fey, J. Eitzinger, G. Hager, and G. Wellein. Analysis of intel's haswell microarchitecture using the ecm model and microbenchmarks. In *International Conference on Architecture of Computing Systems*, pages 210–222. Springer, 2016.

[11] J. Hofmann, G. Hager, G. Wellein, and D. Fey. An analysis of core- and chip-level architectural features in four generations of intel server processors. In *International supercomputing conference*, pages 294–314. Springer, 2017.

[12] S. Lee, J. S. Meredith, and J. S. Vetter. Compass: A framework for automated performance modeling and prediction. In *29*.

[13] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance analysis with cache-aware roofline model in intel advisor. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 898–907. IEEE, 2017.

[14] C. McCurdy, G. Marin, and J. S. Vetter. Characterizing the impact of prefetching on scientific application performance. In *4th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS13)*, Denver, 2013.

[15] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004.

[16] D. Molka, D. Hackenberg, and R. Schöne. Main memory and cache performance of intel sandy bridge and amd bulldozer. In *The workshop on Memory Systems Performance and Correctness*, pages 1–10, 2014.

[17] M. A. H. Monil, M. Belviranli, S. Lee, J. S. Vetter, and A. Malony. Mephesto: Modeling energy-performancein heterogeneous socs and their trade-offs. In *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[18] I. B. Peng, J. S. Vetter, S. V. Moore, and S. Lee. Tuyere: enabling scalable memory workloads for system exploration. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 180–191, Tempe, Arizona, 2018. ACM.

[19] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[20] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[21] M. Umar, S. V. Moore, J. S. Meredith, J. S. Vetter, and K. W. Cameron. Aspen-based performance and energy modeling frameworks. *Journal of Parallel and Distributed Computing*, 120:222–236, 2018.

[22] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[23] L. Yu, D. Li, S. Mittal, and J. S. Vetter. Quantitatively modeling application resiliency with the data vulnerability factor. *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

VIII. Appendix

A. Artifact description

Application (e.g., Vector multiplication) that exhibits streaming and strided memory access pattern is executed on Intel processors from different micro-architectures which include Broadwell, Sky Lake, and Cascade Lake (Detail in Table III). Using TAU and PAPI, uncore counters are measured to find out the traffic between LLC and DRAM. By enabling and disabling prefetching, experimentation is done in all three micro-architectures. Different data structure sizes and initialization factors are also considered. In the end, measured data is compared to predicted data.

B. Artifact availability

Software artifact is available in the github repository(link): MCHPC_AD_AE. There is no author-created hardware artifact. Data artifact is kept in the same github repository and no author-created artifacts are proprietary.

1) Experimental setup: Hardware details are given in Table III. The operating system is Centos-7. GCC 9.1 is used. TAU-2.29 and PAPI-6.0.0.1 are used in measuring LLC-DRAM traffic. PAPI paranoid must be set (see §III-A). Moreover, prefetching enabler/disabler needs to be installed [5].

C. Artifact Evaluation — validation

To facilitate the validation in artifact evaluation, the prediction mechanism is further detailed in this section. The prediction model starts with stream access (i.e., stride = 1). Calculation for streaming access is explained in §II-A.

TABLE IV: Prediction for non-initialized array with or without prefetching.

Stride	Read no pref.	Write no pref.	Read pref.	Write pref.
1	stream	stream	stream	stream
2	stream	stream	stream	stream
4	stream	stream	stream	stream
8	stream	stream	stream	stream
16	stream	stream	stream	stream
32	prev/2	stream	stream	stream
64	prev/1.9	stream	access*3	stream
128	prev/1.8	stream	stream/8	stream
256	prev/1.7	stream	prev/1.7	stream
512	prev/1.5	stream	prev/1.4	stream
1024	prev/1.3	stream	prev/1.3	stream
2048	prev/1.2	stream	prev/1.2	stream
4096	prev/1.1	stream	prev/1.1	stream
8192	prev/1.0	stream	prev/1.0	stream

1) Non-initialized write array with prefetching disabled: The second and third columns of Table IV represent read and write prediction criteria when prefetching is disabled. Since the highest stride is smaller than 2MiB and the array is non-initialized, for write prediction, streaming access data is shown for all strides. For read prediction, streaming access is followed until stride 16 and for stride 32, stream data is divided by two which is denoted by prev/2 (here, prev denotes the data from the previous stride). From stride 64 and onward a ratio is followed which is shown in the table (based on Fig. 4).

2) Non-initialized write array with prefetching enabled: The last two columns of Table IV shows the prediction criteria when prefetching is enabled for non-initialized write array. For write traffic prediction, there is no change when prefetching is disabled. For read traffic prediction, enabling prefetching only impacts stride 32, 64, and 128. For stride 32, when prefetching is enabled, stream access data is followed and for stride 64, total data access (array_size / stride) is multiplied by three since the adjacent cache line and one additional cache line are pulled from DRAM as per prefetching policy. Stride 128 does not have any impact of prefetching so stream data is divided by 8 to follow the regular of no-prefetching.

TABLE V: Prediction for initialized array with or without prefetching.

Stride	Read no pref.	Write no pref.	Read pref.	Write pref.
1	stream	stream	stream	stream
2	stream	stream	stream	stream
4	stream	stream	stream	stream
8	stream	stream	stream	stream
16	stream	stream	stream	stream
32	prev/2	prev/2	stream	prev/2
64	prev/2	prev/2	access*3	prev/2
128	prev/2	prev/2	stream/8	prev/2
256	prev/2	prev/2	prev/2	prev/2
512	prev/2	prev/2	prev/2	prev/2
1024	prev/2	prev/2	prev/2	prev/2
2048	prev/2	prev/2	prev/2	prev/2
4096	prev/2	prev/2	prev/2	prev/2
8192	prev/2	prev/2	prev/2	prev/2

3) Initialized write array with prefetching disabled: The second and third columns of Table V represents read and write prediction criteria when prefetching is disabled. For both read and write traffic prediction, stream data is continued up to stride 16 and then reduces by half when the stride is doubled.

4) Initialized write array with prefetching enabled: Last two columns of Table V shows the prediction criteria when prefetching is enabled for initialized write array. For write traffic prediction there is no change when prefetching is disabled. For read traffic prediction, enabling prefetching only impacts stride 32, 64, and 128 which are predicted similar to non-initialized cases.

5) Data sheet: Data sheet (MCHPC_AD_AE.xlsx) is provided in the repository (MCHPC_AD_AE) which shows prediction values, measured values, and error for each data size for each micro-architecture.

D. Artifact Evaluation — Verification

For artifact verification codes and scripts are given in the repository (link: MCHPC_AD_AE) which are described below:

- [repo]/native_run_and_show_broadwell.sh: to run experiments on Broadwell architecture for all data sizes with prefetching enabled and disabled.
- [repo]/native_run_and_show_skylake.sh: For Sky Lake.
- [repo]/native_run_and_show_cascadelake.sh: For Cascade Lake.
- [repo]/vecmul.c: This is the code for Listing 1

All the figures presented in the paper can be produced by python script in [repo]/graphs location. The scripts are named according to the figure number in the paper.