Symbiotic HW Cache and SW DTLB Prefetching for DRAM/NVM Hybrid Memory

Onkar Patil[1], Frank Mueller[1], Latchesar Ionkov[2], Jason Lee[2], Michael Lang[2]
[1]North Carolina State University [2]Los Alamos National Laboratory
opatil@ncsu.edu mueller@cs.ncsu.edu lionkov,jasonlee,mlang@lanl.gov

***Abstract*—The introduction of NVDIMM memory devices has encouraged the use of DRAM/NVM based hybrid memory systems to increase the memory-per-core ratio in compute nodes and obtain possible energy and cost benefits. However, Non-Volatile Memory (NVM) is slower than DRAM in terms of read/write latency. This difference in performance will adversely affect memory-bound applications. Traditionally, data prefetching at the hardware level has been used to increase the number of cache hits to mitigate performance degradation. However, software (SW) prefetching has not been used effectively to reduce the effects of high memory access latencies. Also, the current cache hierarchy and hardware (HW) prefetching are not optimized for a hybrid memory system.**

We hypothesize that HW and SW prefetching can complement each other in placing data in caches and the Data Translation Look-aside Buffer (DTLB) prior to their references, and by doing so adaptively, highly varying access latencies in a DRAM/NVM hybrid memory system are taken into account. This work contributes an adaptive SW prefetch method based on the characterization of read/write/unroll prefetch distances for NVM and DRAM. Prefetch performance is characterized via custom benchmarks based on STREAM2 specifications in a multicore MPI runtime environment and compared to the performance of the standard SW prefetch pass in GCC. Furthermore, the effects of HW prefetching on kernels executing on hybrid memory system are evaluated. Experimental results indicate that SW prefetching targeted to populate the DTLB results in up to 26% performance improvement when symbiotically used in conjunction with HW prefetching, as opposed to only HW prefetching. Based on our findings, changes to GCC's prefetch-loop-arrays compiler pass are proposed to take advantage of DTLB prefetching in a hybrid memory system for kernels that are frequently used in HPC applications.

***Index Terms*—Prefetching, NVDIMM, Optane DC, DRAM, NVM, Hybrid Memory Architecture, HPC, DTLB**

I. Introduction

Hybrid memory architectures are being increasingly adopted in modern computing systems. Intel's Knights Landing (KNL) introduced High Bandwidth Memory (HBM) along with traditional DRAM-based main memory [1]. General Purpose Graphics Processing Units (GPGPU) are also equipped with HBM [2]. Fujitsu is using a hybrid memory cube (HMC) for its A64FX ARM based chips to deliver high bandwidth memory access to all the compute cores [3]. Recently, Intel launched their Phase Change Memory based Optane DC Persistent Memory Modules (PMM), which are byte-addressable NVDIMMs used as non-volatile main memory (NVM) [4]. The Aurora supercomputer [5], which will be launched in 2021, will have support for Intel Optane DC PMMs. Hence, such a DRAM/NVM based hybrid memory architecture will become more prevalent in future HPC systems. One of the reasons why memory architectures are becoming heterogeneous is because each memory technology brings different characteristics to the table with pros and cons [6]. HBM allows more data to be moved to the processor with the same access latency as DRAM, while NVM provides higher memory capacity supporting application runs with larger problem sizes on fewer systems (nodes), which can result in lower energy consumption and cheaper acquisition costs.

Modern architectures use hardware (HW) prefetchers to increase the amount of cache hits of applications and reduce the effective latency of L1 and L2 cache misses. This can improve overall application performance. HW prefetchers rely on hardware to fetch data by tracking cache line addresses accessed in the N most recently accessed 4KB pages for each core and then predicting future access locations during execution [7], [8]. The HW prefetchers are limited by the 4KB page boundary and so-called huge pages do not alleviate this problem as the HW prefetcher is unaware of the larger page boundary [9]. Also, the number of huge page Data Translation Look-aside Buffer (DTLB) entries are very limited, and their allocation requires specific API calls. Software (SW) prefetching is performed by using the prefetch instructions that are part of the instruction set architecture (ISA) on many processors. Compilers perform SW prefetching by adding prefetch instructions to the code after analyzing memory access patterns statically, which increases the amount of cache hits on the critical path. Each prefetch mechanism has its own advantages and disadvantages. However, currently both mechanisms are designed and fine-tuned for only DRAM-based main memory, and SW prefetching is very conservatively utilized. For instance, the "prefetch-loop-arrays" pass in GCC [10] determines the read and write prefetch distances based on mostly the L1 cache size, DRAM access latency and the 64-byte cache line granularity used for DRAM on Intel architectures. It is very rigid in terms of the heuristics it utilizes to decide if prefetching is profitable or not and explicitly tries to avoid clashing with the HW prefetcher.

SW DTLB prefetching was proposed almost three decades ago [11] but is not implemented in any modern systems or compiler frameworks. The DTLB is an address translation cache that contains entries mapping page numbers to page

This material is based upon work supported by United States Department of Energy National Nuclear Security Administration prime contract #89233218CNA000001 subcontract #508854 dated November 1, 2018 for Los Alamos National Laboratory, NSF grants 1217748 and 1525609 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The unclassified release number is LA-UR-20-27524.

frames. DRAM/NVM based hybrid memory systems aim to support larger datasets for applications, which means more memory pages will be fetched by the CPU [9]. This will create additional pressure on the DTLB because every DTLB-miss is served by a page walk across a 4 or 5 level radix tree, which can incur a high cost on performance at times. Further performance degradation can occur if the page walk results in a page fault and main memory has to be accessed to retrieve page table indirections. NVM access latency can further add to this problem. As we move towards hybrid memory architectures, prefetch mechanisms need to adapt in order to mitigate the performance degradation of applications.

High performance computing (HPC) applications frequently use different solvers, e.g., Partial Differential Equations (PDE), Fast Fourier Transform (FFT) and Conjugate Gradient (CG) [12]. The solvers consist of kernels that perform heavy compute and memory operations on large structured arrays. They perform computations, e.g., using various stencil shapes, which display strong spatial and temporal locality. It is critical to take advantage of the locality of references for good performance of the HPC applications on hybrid memory systems. Upcoming HPC systems will have hybrid memory devices that require a more effective prefetch methodology to achieve good performance.

This paper analyzes the performance of a DRAM/NVM based hybrid memory system for different HPC kernels. Computational kernels are enhanced by SW prefetch instructions and assessed in their relative effect on performance. The objective here is to characterize the prefetch performance for hybrid memory systems while executing HPC workloads so that the current prefetch mechanisms can adapt to new and potentially hybrid memory architectures.

Section II summarizes previous work on hybrid memory systems and SW prefetching. Section III provides an overview of the "prefetch-loop-arrays" pass in GCC and explores the architecture of a hybrid memory system. Section IV presents our experimental setup and the custom benchmark developed in this work for evaluation. Section V presents and discusses results and observations. Section VI proposes modifications to the "prefetch-loop-arrays" pass to achieve SW adaptive prefetching. Section VII summarizes our contributions.

II. Related Work

A number of recent studies have been conducted recently after the launch of Intel's Optane DC PMMs. Yang et al. and Izraelevitz et al. [13], [14] evaluated the read and write memory access characteristics of Optane DC PMM for different file-systems, database applications and performance benchmarks. They found that Optane DC improves the performance of file systems and database applications due to lower latencies than storage devices. Patil et al. [15] characterized the performance of a DRAM/NVM hybrid memory system for HPC applications. They measured the bandwidth performance and energy characteristics of HPC applications runs on Optane DC compared to pure DRAM and DRAM as cache for Optane DC. Peng et al. [16] evaluated Optane DC PMMs in all the configurations available and also measured the performance of separating read and write allocation on a DRAM/NVM memory system. All the above works focus on evaluating the basic performance characteristics of Optane DC under various execution contexts and workloads. Our work primarily focuses of characterizing the effects of prefetching and utilization of the cache in in a byte-addressable DRAM/NVM hybrid memory address space for HPC workloads.

Several works have been conducted on utilizing SW prefetching in order to improve performance on traditional DRAM based memory systems. Callahan et al. [17] were the first to introduce SW prefetching as non-blocking prefetch instruction to eliminate cache miss latency. Mowry et. al [18] introduced and evaluated compiler based SW prefetching that worked in coordination with the HW prefetcher. Bala et al. [11] introduced SW prefetching for DTLB to decrease both the number and cost of kernel DTLB misses. Margaritov et al. [9] proposed a HW-based DTLB prefetch mechanism to reduce the address translation times in modern processor systems. Badawy et al. [19] evaluated the use of SW prefetching and locality optimizations for various HPC workloads for DRAM-based memory systems and found that for some cases SW prefetching has more benefits.

Fuchs et al. [20] designed a HW prefetcher for code block working sets that predicted the future memory accesses of stencil based codes. Swamy et al. [21] introduced a hardware/-software framework to support efficient helper threading on heterogeneous manycores, where the helper thread would perform SW prefetching to achieve higher sequential performance for memory-intensive workloads. Zhao et al. [22] used a dynamic approach to pool memory allocations together and fine tuned the data prefetching based on the access patterns observed in the profiler. Islam et al. [8] evaluated hardware prefetching in a flat-addressable heterogeneous memory comprising HBM and phase change memory (PCM), where a large buffer was placed in HBM to hide the access latency of PCM. Meswani et al. [7] explored various schemes to manage the heterogeneous memory architectures and then refined the mechanisms to address a variety of HW and SW prefetch implementation challenges. Lee et al. [23] evaluated both SW and HW prefetching for various workloads and suggested techniques for cooperative HW/SW prefetching. We aim to utilize DTLB- and cache line-based software prefetching in order to adapt to the upcoming hybrid memory systems with fast and slow access memory devices while improving the performance of HPC workloads.

III. Architecture

A. GCC prefetch-loop-arrays compiler pass

Mowry et al. [18] designed the GCC compiler pass to optimize HPC workloads with SW prefetch hints that work in coordination with the HW prefetcher. This section analyzes the operational characteristics of their prefetch algorithm. The algorithm aims to be fine tuned for DRAM-based memory systems. All constants and heuristics are fixed to values that conform with DRAM specifications, which differ from system to system. The algorithm works on a per loop basis:

1) Gather all memory references in the loop and convert them into a $base + step * iter + delta$ form. Classify them as read or write references and form groups of references based on $base + step$ to identify different access patterns.
2) Calculate the profitability for each memory reference using a heuristic-based cost model that takes into account the total number of instructions, memory references, the miss rate, trip count, prefetch-ahead distance, unroll factor, the maximum prefetch-slots and temporal locality.
3) Determine the references that can be prefetched based on maximum prefetch-slots and prefetch-modulo (relative to the loop iterator).
4) Unroll the loops to satisfy prefetch-modulo and prefetch-before constraints in prologue size, but without loop peeling.
5) Emit the prefetch instructions.

The cost model used to determine the profitability of prefetching for different memory reference groups is as follows:

- First determine the prefetch-modulo and prefetch-before for every memory reference using the $step$ and $delta$ values. Their temporal locality is determined data dependence analysis on the memory references.
- The prefetch-ahead is determined by using the ratio of $\frac{fetch\_time}{iteration\_time}$ indicating how far ahead to prefetch. Both these values are unavailable at compile time. Instead, target-specific constants are used to make an "educated guess".
- The acceptable miss rate is calculated based on the references that have the same $base + step * iter$ but different $delta$. If $delta$ exceeds cache line size then it is determined to be a miss. If the probability of this miss is less than 0.95, then prefetching is considered to be unprofitable.
- It determines if the loop has enough iterations to justify prefetching ahead using the trip-count-to-ahead-ratio with a cut-off threshold of 4 (i.e., no prefetching below this threshold).
- It also calculates the ratio between $\frac{total\_instruction\_count}{memory\_references}$ to determine if the loop has enough CPU instructions to overlap the cache misses. If the ratio is smaller than the machine specific threshold, no prefetching is done.
- It also calculates the prefetch-cost via the ratio of $\frac{total\_prefetch\_count}{total\_instructions\_count}$. If this cost is too high, then no prefetching is performed.
- The ratio of $\frac{prefetch\_mod}{unroll\_factor}$ is calculated with a threshold value of 4, below which no prefetching occurs.

Some of these thresholds are overly strict even by DRAM standards. If the same prefetching parameters were used for NVM memory accesses, the algorithm would fail to gauge the most efficient prefetch configuration or not perform prefetching at all. Although the algorithm acknowledges different types of streams/access patterns in a kernel, it does not consider varying thresholds. Hence, it cannot adapt to different memory access patterns or memory technologies. Also, all parameters are defined based on cache and cache line sizes. No DTLB prefetching is considered.

B. DRAM-NVM hybrid memory architecture platform

The system used in experiments is a single HPE Proliant DL360 node with 2 CPU sockets equipped with Intel's Xeon 8260 (code-named Cascade Lake). Each chip has 24 cores with a clock frequency of 2.4 GHz. Each core has 2 processing units under hyperthreading for a total of 96 CPUs. Each core has a 32 KB private L1 instruction cache, a 32 KB private data cache, and a private 1 MB L2 cache. There is a 35.75 MB L3 cache shared between all cores. It has a DTLB cache with 64 entries, which is 4-way set associative.

Each socket has 12 DIMM slots. 6 of the slots are occupied by 16 GB DDR4 DRAM modules and the other 6 slots are occupied by 128 GB Optane DC modules for a total of 192 GB DRAM and 1.5 TB NVM. The node has 4 memory controllers in total, two are connected to 6 DRAM DIMMs each, and the other two, known as iMC, are connected to 6 NVDIMMs each. The processor uses the standard DDR4 protocol on the regular DRAM memory controller and the DDR-T protocol for Optane DC on the i-memory controller (iMC). Using this proprietary extension of the protocol, the Optane DC achieves asynchronous command/data timing and variable-latency memory transactions. Optane DC has an on-DIMM Apache Pass controller that handles memory access requests and the processing required on NVDIMM. The on-DIMM controller internally translates the addresses of all access requests for wear-leveling and bad-block management. It maintains an address indirection table on-DIMM that translates the DIMM's physical addresses to an internal device address. The table is also backed up on DRAM.

Accessing data on Optane DC occurs after the translation. The controller translates 64 byte load/stores into 256 byte accesses due to the higher cache line access granularity of Optane DC, which causes write amplification [14]. Optane DC PMM can operate in different modes (1) as an uncached byte-addressable memory (Flat mode), (2) as DRAM cached main memory (Memory mode), or (3) as a block storage device (App-Direct mode). All modes (except for Flat) are provided by Intel. Flat is a custom mode introduced by patching the OS kernel to identify all DIMMs as DRAM, thereby creating a true hybrid memory address space. All experiments are performed on the Flat-mode.

IV. Experimental Setup

The aim of this experiment is to characterize the performance of SW prefetching for different prefetch distances under temporal and non-temporal prefetching with allocations on DRAM and NVM separately. We switch HW prefetching on and off for all experiments to evaluate its effect, which requires a reboot after toggling on/off three BIOS setting: HW Prefetch, Adjacent Sector Prefetch, and DCU Stream Prefetch. We compare the SW prefetch performance with and without "prefetch-loop-arrays" compiler optimization of GCC 9.3.0 while using the O3 flag for all compilations. The symbiotic SW prefetching runs are compiled with "no-unroll-loops" in order to measure the effect of varying unroll distances.

We developed a custom benchmark that allows us to measure the prefetch performance for different kernels frequently occurring in HPC applications. These kernels include a write-only (Wr-only) stream, single-write-multiple-read stream (1W4R), and 3-, 5-, 7-, 9- and 27-point stencil streams. The Wr-only stream kernel consists of 5 sequential write streams of linear arrays. The 1W4R kernel has one write stream and four read streams, which are also accessed sequentially. All stencil kernels consist of a write stream and a read stream of a 3-dimensional (3D) dataset of linearly laid out arrays accessed in row-major order using three nested for loops.

The stencil codes are implemented as Jacobi iterative kernels, which are common in Computational Fluid Dynamics (CFD) applications, Partial Differential Equations (PDEs), and pointular automata [24]. Some examples of stencil code-based HPC applications are Vector Particle In Cell (VPIC) [25]–[27] and Algebraic Multi-grid (AMG) [28], which are compute- and memory-bound applications, respectively. The 3-, 5- and 7-point stencils use the Von Neumann neighborhood whereas the 9- and 27-point stencils use the Moore neighborhood [29]. The 3-point stencil is a one-dimensional (1D) stencil, where for every iteration the previous element and the next element are read along with the current one. The 5-point stencil is a two-dimensional (2D) stencil, where along with adjacent elements in the same row of the current element, adjacent elements in the same column of the current element are also read. The 7-point stencil is a 3D stencil, where along with the adjacent elements in the same row and column of the current element, adjacent elements in the next and previous plane are read. The 9-point stencil is a 2D stencil including diagonal elements beyond the 5-point stencil. Similarly, 27-point stencil is a 3D stencil with diagonals on every dimensional pair beyond the 7-point stencil. These stencils comprise one or more read streams, plus a write stream accessed sequentially. Each stream is 4 GB in size and is allocated separately on each NUMA node using numa_alloc_onnode() for every run.

We manually unroll and peel the kernels. Each kernel has a prologue and an epilogue loop. The prologue loop prefetches each element of the stream sequentially until up to given read or write prefetch distance. The main compute kernel is unrolled up to the unroll distance and the next elements are prefetched after unrolling is complete. The main loop stops when there are no more elements to prefetch and the remaining iterations are completed in the epilogue loop. Due to the variability in the read and write distances, the prologue and epilogue loops are split and separated by conditional statements to avoid over-prefetching and segmentation faults.

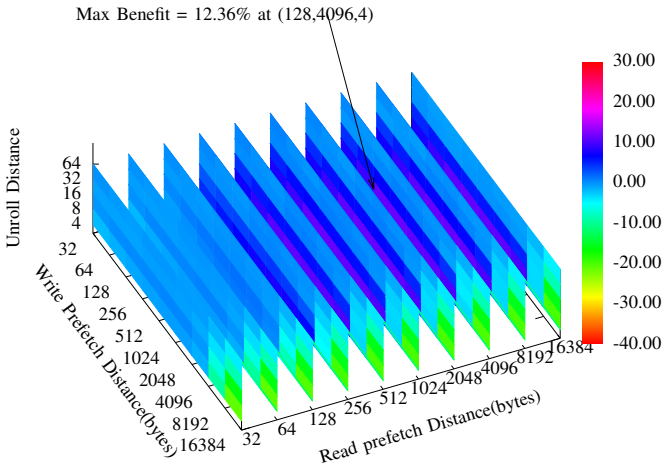
We use the GCC __builtin_prefetch() function to prefetch the desired cache line at every iteration [30], which automatically calls the corresponding intrinsic for a given instruction set architecture. We change the read and write prefetch distance explicitly from 32 bytes to 16,384 bytes using nested loops that encompass all the kernels. The upper bound of the prefetch distance is kept at 16,384 bytes to avoid L1 cache contention at higher distances. We change the parameters of the prefetch call to perform non-temporal and temporal prefetching for linear and stencil read streams, respectively. We use non-temporal prefetching for all write streams. For non-temporal prefetching, the data is fetched into a non-temporal cache structure to avoid cache pollution; whereas for temporal prefetching the data is prefetch into L2 cache or higher [31], [32]. We also vary the unroll distance in another nested loop from 4 to 64. We limit the unroll distance to 64 to restrict additional pressure on available CPU registers. We perform cache line prefetching with distances from 32 to 2,048 and perform DTLB page prefetching from distances from 4,096 to 16,384. This is enabled by adding a conditional statement to the prefetch statement block, which prefetches only after certain number of iterations have elapsed nearing the page boundary. DTLB caching causes the address translation to be stored in the 4-tier DTLB cache, which reduces future page faults that are expensive in terms of CPU cycles [33].

We refer to our technique as "symbiotic prefetching" from here on. We execute the benchmark on 48 processes running individually on each core launched by MPI, but without imposing communication via message passing, i.e., just to provide realistic workloads in separate address spaces with contention on shared resources (last-level cache, DRAM/NVM). We divide the stream size equally between all processes and allocate them separately. We pin all processes to the cores and then calculate the data bandwidth for DRAM and NVM allocations using the ratio of total memory of all data structures accessed and wall clock time measured for each kernel. Each measurement is averaged over 10 runs, where a standard deviation of 4%-6% is observed for all kernels. To obtain cache performance metrics, we use the LIKWID Marker API [34] to measure metrics for every individual kernel obtained from HW performance counters. The performance counters are obtained for the configuration that delivers highest performance benefit and then reported relative to measurements of the same kernel without any prefetching as a baseline. We also perform the same comparison for the kernels when they are compiled using "prefetch-loop-arrays". We term this as compiler prefetching from here on.

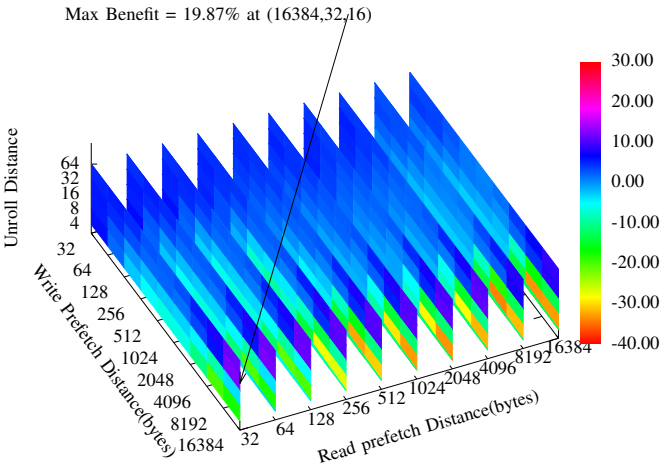
V. Results

This section discusses the results of experiments and present observations. Prefetch performance is depicted as percentage changes over different compilation options. We depict percentage changes in data bandwidth observed with symbiotic prefetching (sp) as a heat map over a 3D graph relative to the baseline bandwidth observed when compiled with no prefetching (np). The x-axis depicts the write prefetch distance in bytes, the y-axis the read prefetch distance in bytes, and the z-axis the unroll distance in number of iterations. The heatmap colors represent percentage changes in bandwidth relative to the baseline. We plot the graphs for DRAM (left) and NVM (right) separately.

We only present heatmaps for the 7-point stencil in Figures 1 and 2 due to space limits, but we report and discuss the results (data bandwidth and performance metrics) for all kernels as all have similar graphs, albeit with different best values (subject of a forthcoming technical report). Figure 1 depicts



(a) DRAM sp % change



(b) NVM sp % change

Fig. 1: Performance of 7-pt stencil (3D) stream with Temporal prefetching relative to no HW prefetching as a baseline

results of symbiotic and compiler prefetching for the 7-point stencil kernel without HW prefetching. Figure 1a and 1b depict performance changes under symbiotic prefetching for DRAM and NVM, respectively, where an arrow indicates the highest benefit configuration (write, read, unroll distances).

Observation 1: For DRAM, bandwidth increases for larger read prefetch distances (x-axis) but abruptly drops close to the largest write prefetch distances (y-axis) and is best for a small unroll distance (z-axis).

Observation 2: For NVM, the bandwidth decreases toward higher write and read prefetch distances (x+y axes) and, in contrast to DRAM, is best for largest unroll distance (z-axis).

Observation 3: For DRAM, symbiotic prefetching results in slightly higher L1-DTLB load misses, and even slightly more for compiler prefetching. For NVM, L1-DTLB load misses are significantly higher by a factor of 11x (1100%) under symbiotic prefetching compared to no SW prefetching — where compiler prefetching actually *reduces* the L1-DTLB load misses for NVM. Symbiotic prefetching also reduces the L1-DTLB store misses for NVM whereas compiler prefetching increases them.

Given the structure of a 3D 7-point stencil, a total of 5 read streams and 1 write stream exist. Hence, performance of the kernel critically depends on the availability of read data in the cache. As the stencil moves across the data set, there is reuse of all neighboring read data points. Hence, it is prefetched as temporal data into the L2 and L1 caches. The write data points do not have reuse, and are hence prefetched as non-temporal data into a separate cache structure. However, if the latency of the memory accesses is large then a high unroll distance is required to reduce CPU stall cycles. The dependence on unroll distance is reflected in the NVM heatmap where the performance benefits from higher unroll distances, which increases temporal reuse of read data and overlaps prefetch latency of the write stream with computation. Nonetheless, a short unroll distance is sufficient for faster DRAM memory accesses under DTLB prefetching for the read streams. The pages for read data are quickly cached into the DTLB, which increases the reuse within read streams by reducing page walks and page faults. Further, short unroll distances are sufficient to overlap with prefetches spaced according to the write stream. This is also reflected in hardware counter metrics, where a reduction in L2, L3 and L1-DTLB store misses is observed for symbiotic prefetching on DRAM, which is the source of the performance benefit. For NVM, the L1-DTLB load misses are high due to a smaller read prefetch distance, and the reduction in store misses provides the main performance benefit. Cache and memory bandwidths increase as a result of reduced L1-DTLB misses.

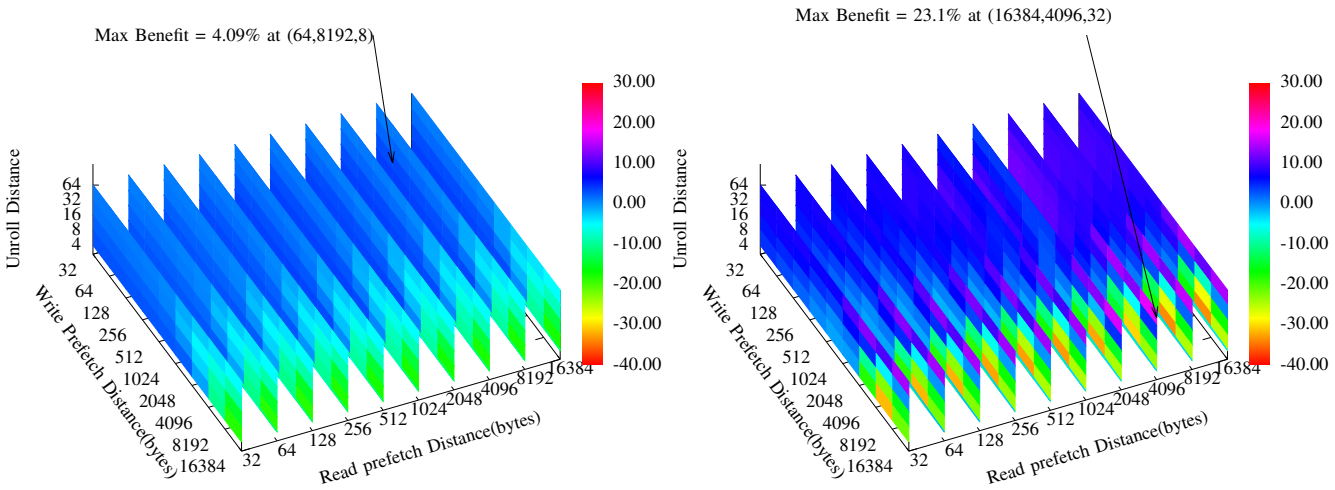
Let us also consider Figure 3a in this context, which depicts the performance of all prefetching methods relative to no prefetching as a baseline per kernel; and Figure 3b, which depicts the performance comparison of HW prefetching relative to no HW prefetching as a baseline. The y-axis depicts the percentage change in data bandwidth and the x-axis lists all benchmark kernels for both figures. For the 7-point stencil, compiler prefetching is not able to provide any performance benefit as seen in Figure 3a whereas symbiotic prefetching on the other hand provides a 12% and 19% performance benefit for DRAM and NVM, respectively, over no prefetching. This is also reflected in the hardware counters, where compiler prefetching shows the smallest changes over these metrics. This results from tight bounds plus low margins on heuristics and greater dependence on HW prefetching, where its absence harms performance.

Inference 1: *The prefetching configurations show diametric behavior for the same kernel when its streams are allocated on DRAM and NVM. SW prefetching provides benefits for both DRAM and NVM without relying on or being complemented by the HW prefetcher.*

Figure 2 depicts the results for symbiotic and compiler prefetching for the 7-point stencil kernel with HW prefetching, with the same subfigures as before.

Observation 4: For DRAM, the bandwidth increases toward lower write and also slightly toward higher read prefetch distances, and it slightly increases for smaller unroll distances.

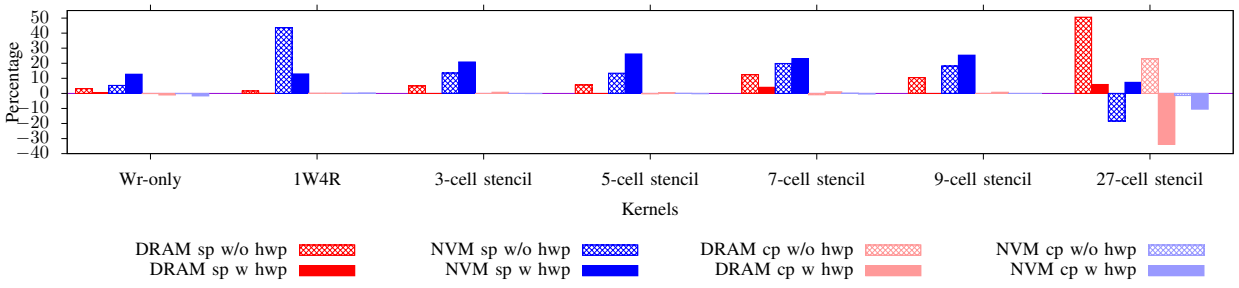
Observation 5: For NVM, the data bandwidth increases as



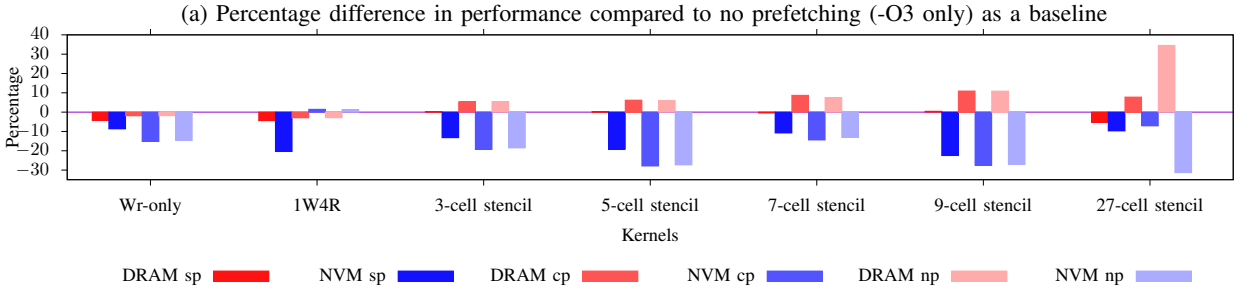
(a) DRAM sp % change

(b) NVM sp % change

Fig. 2: Performance of 7-pt stencil (3D) stream with L2 Temporal prefetching relative to HW prefetching as a baseline



(a) Percentage difference in performance compared to no prefetching (-O3 only) as a baseline



(b) Percentage difference in performance with HW prefetching compared to no HW prefetching as a baseline

Fig. 3: Performance Comparison between all prefetching methods (sp = symbiotic prefetching, cp = Compiler prefetching, np = no prefetching, hwp = HW prefetching)

we move higher on all three axes.

Observation 6: L1-DTLB load misses for DRAM increase under symbiotic prefetching but decrease under compiler prefetching. L1-DTLB store misses are reduced by symbiotic prefetching, whereas compiler prefetching increases it. A similar but more profound value change is observed for NVM.

As symbiotic prefetching operates independently of HW prefetching, hardware counters change according to bandwidth trends of Figures 2a and 2b. The increase in darkness on the heatmap indicates that HW and SW prefetching *together* are able to provide the best performance benefit. For DRAM, additional HW prefetching does not significantly affect symbiotic prefetching but complements its performance by enabling higher read prefetch and unroll distances to better take advantage of temporal locality for this 7-point stencil kernel. Every dispatched load operation by the HW prefetcher is first dispatched to the L1 cache and DTLB. Upon an L1 miss, it is relayed to lower caches, which can incur up to 80 cycles to be fetched. However, a TLB miss can result in a page walk or page fault accounting for thousands of cycles. However, due to the DTLB prefetching, the dispatched load encounters a TLB hit most of the time reducing the overhead of page walks and page faults.

Nonetheless, the addition of HW prefetching provides benefits (see Figure 3a). NVM is also assisted by HW prefetching with larger prefetch and unroll distances for symbiotic prefetching compared to no HW prefetching. SW read prefetching affects the DTLB cache, which results in fewer L1-DTLB load misses relative to the baseline (only HW prefetching). Here, HW prefetching serves the temporal locality of the access pattern. This frees up the SW prefetcher, which is now utilized to reduce TLB misses. Hence, the performance

benefit of symbiotic plus HW prefetching results in twice the performance gain compared to no HW prefetching for NVM (see Figure 3a). Compiler prefetching cannot improve performance by much without DTLB prefetching; it is DTLB prefetching that provides the main source of improvement (see heatmaps of DRAM and NVM).

Inference 2: *SW prefetching improves DTLB performance in a manner symbiotic to HW prefetching driving cache benefits for the kernel for both DRAM and NVM.*

Any requests of the HW prefetcher benefit upper caches while symbiotic prefetching effectively becomes DTLB prefetching and results in reduced L1-DTLB load misses. Although this is observed for NVM, symbiotic prefetching is not able to increase the L1-DTLB load misses for DRAM, as the latter (DRAM) does not benefit in performance. Due to fewer read streams, HW prefetch requests populate the DTLB cache quickly enough that symbiotic prefetching becomes redundant. The heatmaps of all the kernels show a similar behavior with subtle differences owing to the number of read and write streams present in the kernel. The SW prefetch configurations indicated by the arrows that deliver the highest performance benefit for each kernel are summarized in Table I. The table reinforces the inference that SW prefetching should be utilized alongside HW prefetching to combine DTLB caching and data caching due to the symbiosis between former and latter prefetch techniques, respectively. But configurations have to be adapted to the underlying memory technology (DRAM vs. NVM) and specific access patterns.

Observation 7: We observe that symbiotic prefetching by itself provides more performance benefit than compiler prefetching for DRAM and NVM over all kernels. The highest performance benefit is observed when symbiotic prefetching is used in conjunction with HW prefetching from 4 to 26%. The performance for NVM degrades when combined with HW prefetching for all kernels, except for 1W4R. But notably, symbiotic prefetching mitigates this degradation (except for the rather simplistic Wr-only and 1W4R kernels). In contrast, compiler prefetching remains ineffective, i.e., any degraded performance cannot be reduced. The HW prefetcher degrades the performance of linear array streams on DRAM, and SW prefetching is unable to mitigate this problem.

Inference 3: *HW prefetching only serves short-distance read prefetches that have high temporal locality, which are typical signatures of 1D or 2D stencil kernels. HW prefetching also needs to be adaptive to access patterns in order avoid performance degradation. Here, SW prefetching comes to the rescue as it mitigates these performance degradations on NVM.*

VI. Adaptive SW prefetching as a compiler pass

Based on our observations and inferences, we propose the following changes to the "prefetch-loop-arrays" compiler pass in GCC:

- To adapt to a hybrid memory, compilers should become memory allocation-aware. This can be accomplished by overloading malloc() where the programmer along with the size can also specify the desired memory device and the NUMA nodes need to be mapped to the correct memory device to complete the allocation.
- The pass already identifies different streams and their access patterns in any given loop. Hence, classifying specific workloads is feasible and should be incorporated.
- Instead of a single set of constants and threshold values to guide heuristics and the cost model, a static table for each memory technology should be maintained with specific constants/thresholds per access pattern. Our contributions in this work lay the foundation to automatically derive these constants and thresholds in a calibration run, which then allows the derivation of values similar to Table I. Once access pattern and memory type of streams have been determined, the compiler pass can readily decide on prefetches given the specific constants/thresholds.
- Condition check for non-temporal locality should be removed, and non-temporal prefetches should be supported. Checking of the HW prefetcher stride needs to be lifted to allow for both symbiotic SW and HW prefetching.
- The acceptable miss rates need to be lowered for large distances to accommodate DTLB prefetching. Similarly, many thresholds (trip-count-to-ahead-ratio, prefetch-mod-to-unroll-factor-ratio, memory-ref-count-reasonable and insn-to-prefetch-ratio-too-small also) need to be higher for symbiotic prefetching to allow DTLB prefetching in software — as well as support for higher unroll distances that are adaptive for slow and fast memories.
- With predictable access patterns, priority ordering of prefetches based on lowest prefetch-modulo can be replaced by the (more easily) predicted performance benefit.
- Finally, loop peeling is required when emitting prefetch instructions at the end of the pass.

The changes are beyond the scope of this work, but only become feasible due to the contributions of our memory-hybrid, adaptive and symbiotic prefetching.

VII. Conclusion

Our work provides novel insight that the existing rigid and conservative approach to SW prefetching leaves ample performance potential on the table for HPC workloads. We show that existing HW prefetchers are neither optimized for NVM memory nor for non-temporal workloads. We contribute HW and SW prefetch methods that are more adaptive and show that they succeed in extracting symbiotic performance while being sensitive to hybrid memory systems. Our DTLB-based symbiotic SW prefetching improves the performance of HPC kernels from 4 to 26% for data streams allocated on both DRAM and NVM, and our SW prefetching complements HW prefetching rather than competing with it. We also present a simple design to modify an existing SW prefetch compiler pass to implement our prefetch configurations with the potential to automatically improve performance for HPC workloads on future hybrid memory systems.

References

[1] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-

TABLE I: Symbiotic prefetching configurations and performance benefits

Kernel	Prefetch Distances(bytes), Unroll length and Performance Improvement(%)															
	DRAM								NVM							
	With HW prefetch				Without HW prefetch				With HW prefetch				Without HW prefetch			
	Read	Write	Unroll	%	Read	Write	Unroll	%	Read	Write	Unroll	%	Read	Write	Unroll	%
Write-only	-	256	64	0.6	-	128	32	3.13	-	4096	32	12.70	-	4096	32	5.27
1W4R	128	128	64	0.02	128	1024	32	1.57	16384	4096	32	12.93	16384	8192	4	43.62
3-pt stencil	64	64	64	-0.07	512	128	4	4.94	4096	16384	32	20.83	4096	16384	16	13.49
5-pt stencil	64	64	64	-0.01	16384	512	8	5.62	4096	16384	64	26.15	32	16384	32	13.41
7-pt stencil	8192	64	8	4.09	4096	128	4	12.36	4096	16384	32	23.1	32	16384	16	19.87
9-pt stencil	1024	32	64	-0.11	2048	256	8	10.47	4096	16384	64	25.38	32	16384	32	18.12
27-pt stencil	4096	64	32	5.89	16384	128	8	50.6	2048	16384	64	7.33	64	4096	16	-18.36

generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[2] M. O'Connor, "Highlights of the high-bandwidth memory (hbm) standard," in *Memory Forum Workshop*, 2014.

[3] Fujitsu hpc roadmap. [Online]. Available: https://www.fujitsu.com/global/Images/fujitsu-hpc-roadmap-beyond-petascale-computing.pdf

[4] J. Hruska, "Intel announces new optane dc persistent memory," *ExtremeTech https://www. extremetech. com/extreme/270270-intel-announces-new-optane-dc-persistent-memory*, 2018.

[5] Aurora - may 2019. https://press3.mcs.anl.gov/aurora/.

[6] O. Mutlu, "Memory scaling: A systems architecture perspective," in *2013 5th IEEE International Memory Workshop*. IEEE, 2013, pp. 21–25.

[7] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 126–136.

[8] M. Islam, K. M. Kavi, M. Meswani, S. Banerjee, and N. Jayasena, "Hbm-resident prefetching for heterogeneous memory system," in *International Conference on Architecture of Computing Systems*. Springer, 2017, pp. 124–136.

[9] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1023–1036.

[10] R. M. Stallman, "Gnu compiler collection internals," *Free Software Foundation*, 2002.

[11] K. Bala, M. F. Kaashoek, and W. E. Weihl, "Software prefetching and caching for translation lookaside buffers," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1994, p. 18.

[12] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The hpc challenge (hpcc) benchmark suite," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, vol. 213, no. 10.1145. Citeseer, 2006, pp. 1 188 455–1 188 677.

[13] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[14] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[15] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules," in *Proceedings of the fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE, 2019, pp. 288–303.

[16] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the fifth ACM/IEEE International Symposium on Memory Systems*. ACM/IEEE, 2019, pp. 304–315.

[17] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[18] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *ACM Sigplan Notices*, vol. 27, no. 9, pp. 62–73, 1992.

[19] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng, "The efficacy of software prefetching and locality optimizations on future memory systems," *Journal of Instruction-Level Parallelism*, vol. 6, no. 7, 2004.

[20] A. Fuchs, S. Mannor, U. Weiser, and Y. Etsion, "Loop-aware memory prefetching using code block working sets," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 533–544.

[21] B. N. Swamy, A. Ketterlin, and A. Seznec, "Hardware/software helper thread prefetching on heterogeneous many cores," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 214–221.

[22] Q. Zhao, R. Rabbah, and W.-F. Wong, "Dynamic memory optimization using pool allocation and prefetching," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 27–32, 2005.

[23] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 1, pp. 1–29, 2012.

[24] R. de la Cruz and M. Araya-Polo, "Modeling stencil computations on modern hpc architectures," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 149–171.

[25] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, 2008. [Online]. Available: https://doi.org/10.1063/1.2840133

[26] K. J. Bowers, B. J. Albright, L. Yin, W. Daughton, V. Roytershteyn, B. Bergen, and T. J. T. Kwan, "Advances in petascale kinetic plasma simulation with VPIC and roadrunner," *Journal of Physics: Conference Series*, vol. 180, p. 012055, jul 2009. [Online]. Available: https://doi.org/10.1088\%2F1742-6596\%2F180\%2F1\%2F012055

[27] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson, "0.374 pflop/s trillion-particle kinetic modeling of laser plasma interaction on roadrunner," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 63:1–63:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413435

[28] U. M. Yang *et al.*, "Boomeramg: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[29] T. Kretz and M. Schreckenberg, "Moore and more and symmetry," in *Pedestrian and evacuation dynamics 2005*. Springer, 2007, pp. 297–308.

[30] Gcc data prefetching support. [Online]. Available: https://gcc.gnu.org/projects/prefetch.html

[31] Intel® 64 and ia-32 architectures optimization reference manual. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf?countrylabel=Colombia

[32] Intel® 64 and ia-32 architectures software developer's manual. [Online]. Available: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4

[33] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[34] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.