MDACache: Caching for Multi-Dimensional-Access Memories

Sumitha George*[1] Minli Julie Liao*[1] Huaipan Jiang[1] Jagadish B. Kotra[1,2]
Mahmut T. Kandemir[1] Jack Sampson[1] Vijaykrishnan Narayanan[1]

[1]*Pennsylvania State University*
State College, PA, USA
{sug241, mjl5868, hzj5142, kandemir, sampson, vijay}@cse.psu.edu

[2]*AMD Research* (Work done while at Penn State)
Austin, TX, USA
Jagadish.Kotra@amd.com

***Abstract*—For several emerging memory technologies, a natural formulation of memory arrays (cross-point) provides nearly symmetric access costs along multiple (e.g., both row and column) dimensions in contrast to the row-oriented nature of most DRAM and SRAM implementations, producing a Multi-Dimensional-Access (MDA) memory. While MDA memories can directly support applications with both row and column preferences, most modern processors do not directly access either the rows or columns of memories: memory accesses proceed through a cache hierarchy that abstracts many of the physical features that supply the aforementioned symmetry. To reap the full benefits of MDA memories, a co-design approach must occur across software memory layout, the mapping between the physical and logical organization of the memory arrays, and the cache hierarchy itself in order to efficiently express, convey, and exploit multi-dimensional access patterns.**

In this paper, we describe a taxonomy for different ways of connecting row and column preferences at the application level to an MDA memory through an MDA cache hierarchy and explore specific implementations for the most plausible design points. We extend vectorization support at the compiler level to provide the necessary information to extract preferences and provide compatible memory layouts, and evaluate the tradeoffs among multiple cache designs for the MDA memory systems. Our results indicate that both logically 2-D caching using physically 1-D SRAM structures and on-chip physically 2-D caches can both provide significant improvements in performance over a traditional cache system interfacing with an MDA memory, reducing execution time by 72% and 65%, respectively. We then explore the sensitivity of these benefits as a function of the working-set to cache capacity ratio as well as to MDA technology assumptions.

***Index Terms*—Cache design, Symmetric memories, Crosspoint memory, Row-Column vectorization, 2D memory access**

I. Introduction

Across myriad algorithms spanning from matrix multiplication to vision processing to database queries, many fundamental problems in memory access optimization, in both software and hardware, stem from the disparities between a one-dimensional, linearized model of memory and data structures and algorithms that naturally organize data and data traversals along multiple dimensions. In fact, many existing data layout optimizations try to reorganize the data within program structures to maximize the occurrence of unit-stride accesses on a linear memory space [1]–[6]. However, these disparities arise not only because of the logically one-dimensional nature of the abstract memory model employed, but also because the internal physical organization of data arrays for the currently dominant memory technologies (e.g., SRAM and DRAM) provide strongly asymmetric accesses to row and non-row patterns. Transfers between levels of the cache and memory hierarchy are likewise designed in a row-oriented fashion, aligned along the same common access dimension. Thus, in current systems, several key metrics, from the utilization efficiency of data fetch bandwidth to the ability to leverage SIMD vectorization, are all dependent on mapping both data and accesses into the same, linearized dimension.

Intriguingly, the rise of *crosspoint memories* as a natural array structure for several emerging nonvolatile memory technologies fundamentally alters the physical asymmetry considerations that currently limit multidimensional memory accesses. Collectively, memory technologies, such as PCM, STT-MRAM, and ReRAM, that utilize resistance to store values can be used to construct cells where values can be read or written along *both* the horizontal and vertical axes of a crosspoint array [7] or similar structures [8]. Crosspoint organizations for memory rely on sensing the presence of a conducting path between the horizontal and vertical interconnects through the cell at the crossing point between the two axes rather than relying on the cell itself as a source of charge or current to drive the memory output, which can lead to substantially higher memory density and naturally synergizes with passive (non-volatile) memory storage cells. We term memories that support data transfer along both the horizontal and vertical axes of such topologies **Multi-Dimensional-Access** (*MDA*) **Memories**, and several recent works have already explored the benefits of exploiting crosspoint array features within main memory modules, including in the analog domain, to improve application performance [9]–[11]. However, while MDA memories can provide enhanced support for *both* row and column accesses, the current software view of memory is still fully *linearized,* current caching systems remain optimized for that linear view of memory. The existing cache hierarchies decouple processor address sequences from multi-dimensional array accesses, and implementing MDA memories on-chip faces a distinct set of technology challenges from MDA main

This work is supported in part by NSF grants 1317560,1822923, 1439021, 1629915, 1626251, 1629129, 1763681, 1526750, 1439057,1500848 and Semiconductor Research Corporation JUMP CRISP.
*Equal contribution by first & second author.

memory designs.

In this paper, we propose a set of design changes to the cache hierarchy to construct caches for MDA memories (*MDA-Caches*) that can better express and exploit multi-dimensional application preferences throughout the entire cache and memory hierarchy. We describe a taxonomy for different ways of connecting row and column preference at the application level to an MDA memory through the MDA cache hierarchy and explore specific implementations for the most plausible design points. One key observation is that providing dense memory accesses along both row and column orientations greatly eases *vectorization* along both row and columns. Moreover, in cases where it is unclear how to choose which loop ordering will provide a better optimization or the compiler cannot reconcile multiple conflicting access patterns to the same data structure, supporting both row and column accesses can simplify (or even obviate the need for) some ambiguous compiler tradeoffs.

This work focuses on the design space of MDA caching approaches that can simultaneously hold both row and column aligned data. In a physically 1-D cache, this introduces new challenges in tag management, potentially duplicated data words due to intersecting rows and columns residing in the same cache level, and opens up policy spaces for dealing with new phenomenon such as cache hits with mis-oriented access preference and 2-D MSHR miss coalescing. Physically 2-D caches, where the memory arrays used to implement the cache are themselves MDA memories, remove challenges with data duplication, but require a 2-D block as the fundamental *allocation unit* in a cache. We explore practical implementations of both logically 2-D caches built atop physically 1-D SRAM and caches with both logical and physical 2-D properties (implemented in STTRAM) and demonstrate substantive bandwidth, vectorization, and performance benefits over a baseline 1-D logical/physical design. Given the large transfer unit of a 2-D block in a physically 2D cache, we elide consideration of "dense" 2-D blocks, and directly explore a variant that supports "sparse" occupancy for physically 2-D caches.

While the proposed changes relative to traditional designs are primarily in the cache hierarchy, some buy-in from both the compiler and the OS is necessary to fully exploit the hardware support for multidimensional access. Specifically, the memory layout must align logical and physical dimensions within the MDA, and supporting physical column accesses requires changes to the OS page allocation to ensure that all of the elements in a column (which may span multiple pages) are part of the same process. Fortunately, while these changes are required in order for applications to benefit from MDA memories, they are relatively straightforward to implement for some codes, such as linear algebra operations over matrices, that are positioned to be among the most direct beneficiaries of MDA memories. We describe the software support necessary for MDA memories in Section V.

The specific contributions of this work include:

• We propose "MDA caching" as a means to better exploit the emerging class of MDA memories. We show that, to take full advantage of the MDA memory benefits, co-design must occur across software memory layout, the mapping between the physical and logical organization of the memory arrays, and the cache hierarchy itself, in order to efficiently express, convey, and exploit multi-dimensional access patterns.

• We describe a taxonomy for MDA caching along logical and physical dimensions, discuss the nature of interfaces between different cache hierarchy levels belonging to different taxonomy groups, and describe techniques for implementing logically 2-D MDA caching using both physically 1-D and physically 2-D approaches.

• We observe that supporting column access opens up new opportunities for vectorization support to exploit MDA access, and analyze the degree to which mixed row and column vectorization opportunities are present in a variety of kernels.

• We quantify the benefits supplied by MDA caches over a traditional caching approach when both are attached to an MDA main memory. We discuss the sources of these benefits, showing that MDA caching provides superior bandwidth utilization compared to prefetching, in addition to enhanced vectorization opportunities. Our results indicate that logically 2-D caching using physically 1-D SRAM structures provides 72% average reduction in execution time over a traditional cache system interfacing with an MDA memory. We also show that utilizing an MDA memory technology on-chip to implement a cache that is both logically and physically 2-D can provide 65% reduction benefits once sparsity optimizations are applied.

• Further, we perform sensitivity experiments to examine the degree to which these results still hold for entirely "cache-resident" working sets as well as across differing assumptions on the properties of MDA memory and cache implementation technologies. We demonstrate that, while the approach is sensitive to these dimensions, the benefits remain consistently positive.

II. Overview of Enabling Technologies

The current landscape of competing emerging memory technologies is quite diverse: STTRAM uses MTJs to store information in the direction of the magnetic layer [12]; PCM uses differential resistance between crystalline and amorphous phases of a material [13]; FeRAM uses remnant polarization in a ferroelectric layer [14]; and ReRAMs use the presence/number of filaments to represent bits. Despite this variety, the chief desirable features offered by these memories are quite similar. For example, STTRAM, PCRAM, and ReRAM all aim to provide high density memories that can dramatically increase storage per unit area [12], [13], [15]. In addition to the small device count and feature scaling of the memory storage elements themselves, a key means for all of these technologies in achieving high density at memory array scale is that they are compatible with *crosspoint memory topologies*. Crosspoint memory arrays using STTRAM, ReRAM, PCM, and other resistive memory elements have been thoroughly described in the research literature and embraced by industry [16]–[18]. The most important features of a crosspoint architecture, from the perspective of designing an MDA memory, can be

generalized beyond resistive memory elements: for example, emerging memory cells built from FerroElectric FET (*FeFET*) devices are "symmetric" in structure across drain and source, which has been demonstrated to provide the flexibility of interchanging the read directions [8]. While transpose access designs for existing technologies have been demonstrated and can also provide performance benefits, they come with significant area overheads [19], in stark contrast to crosspoint arrays for these emerging resistive memories, for which crosspoints are a naturally dense organization. Furthermore, the crosspoint organization for resistive memories has enabled analog inner-product computations that have shown substantial benefits in implementing synapse functionality in neural networks [9], [10], [20], [21] and recent commercial products have utilized 3-D crosspoint organizations for nonvolatile memory/storage [22]. Thus, it is expected that many future memories will be organized around a topology that fundamentally eases the implementation of multidimensional accesses. Nonvolatile memories have been integrated at different layers of the memory hierarchy including caches [23], [24], memory, and solid state storage [22]. The device parameters and circuit structure can be tuned to meet the desired tradeoffs between density and speed, resulting in configurations with diverse timing, power, persistence, and even read-write asymmetry tradeoffs [25]. In the scope of this work, we presume an STT-based NVM (with timing parameters modeled on Everspin [26] devices) as the main memory rather than some commercial offerings that have the NVM as a secondary memory fronted by DRAM. To account for technological uncertainty in the relative performances of future NVM memories versus future traditional memories, in Section VIII we examine both the sensitivity to and relative performance of our approach against a faster main memory. In the same section, we also explore the impact of increasing the read-write asymmetry for on-chip MDA memory technologies.

While each of these emerging technologies has its own unique tradeoffs, the work in this paper relies only on common properties shared by all of the above technologies. A general schematic structure of a crosspoint array, similar to those considered in prior proposals [7], [9], is given in Fig. 1. One significant development in realizing MDA memories from crosspoint memory arrays are bidirectional array selectors [27]. Bidirectional selectors add the capability in the crosspoint memory to read and write along both of the crossing interconnect directions. In addition to being able to swap the role of word line and bit line, duplication or alteration to sensing and driving circuitry is also required.

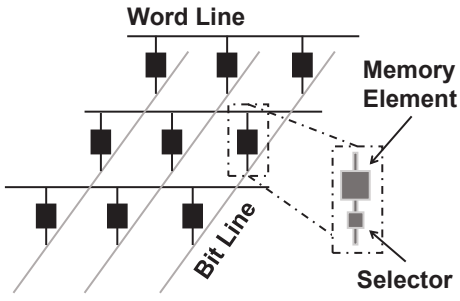


Fig. 1. A general schematic of a crosspoint memory.

In the remainder of this work, we consider using STTRAM to implement physically 2-D memories, although we believe that our approach directly extends to other emerging technologies deployed in crosspoint topologies. STTRAM was chosen as the technology for our MDA memory exploration because we aim to explore physically 2-D implementations of on-chip cache memories as well as off-chip MDA memories, and wish to utilize a single base technology for both on and off-chip 2-D memories in order to preserve clarity on the microarchitectural versus technology contributions to MDA caching approach benefits. STTRAM simplifies some idiosyncrasies of other candidate resistive crosspoint technologies, specifically, endurance management, when considering last-level cache (LLC) implementations [28]. In the next section, we describe how to map from physically multi-dimensional access to logical memory semantics aligned with both horizontal and vertical read/write accesses. We propose a new organization structure to the main memory and cache, and incorporate software support to enable the multi-direction-mode data transfers.

III. From Mats to Memories

This section introduces the proposed memory system. Here, we extend the bidirectional data transfer of the underlying memory circuits to a logical memory organized around bidirectional data transfer capabilities matched to the address space semantics. Our architecture supports two modes of transfer: *row mode* and *column mode*. In row mode, the memory provides a set of data words with unit stride, and in column mode the memory provides the same quantity of data words with a fixed non-unit stride. The chief challenge in leveraging underlying circuit symmetry is that the symmetry is at *bit-level*, rather than *word-level*. Seen from the same dimensionality perspective motivating MDA memories, our challenge is that we are embedding 3-dimensional semantics (row, column, word) for bits into a 2-D (row, column) physical substrate. Thus, to get columns of words in an access, we will need to interleave the bits of the word, which is discussed below. The proposed 2D memory is shown in Fig. 2. Fig. 2(a) shows a traditional memory, where the data is transferred to the row-buffer by opening the rows of the memory array. In this architecture, transferring a column of data column-wise requires multiple row openings. Note that row opening is a costly operation for a memory array in terms of both latency and power. Fig. 2(b) shows an architecture where this particular issue is addressed by introducing a *column buffer* for column-wise transfers. The peripheral circuit for such a 2-D memory includes an extra set of decoders and sensors for the column, and we need to make the memory controller "aware" of the column sensing (Fig.3) [29]. The area overhead of having the extra set of decoders is typically less than 1% [30].

The logical, word-level organization of the 2D main memory is given in Fig. 4. From Fig. 4, for a request mapping to row R1, we expect to see the data X11,X12–X18 transfered to the row-buffer from the main memory array. Similarly, in a column mode request for C1, we would see the data X11,X21–X81 transferred to the column-buffer and then to the

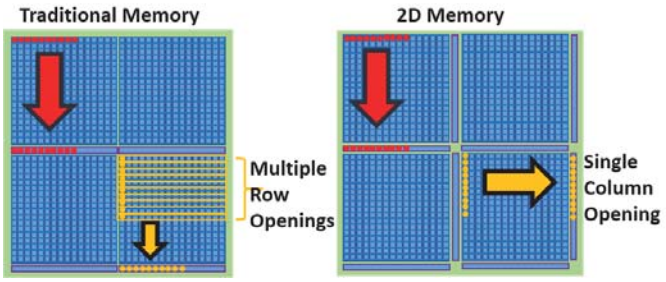


Fig. 2. Main Memory

memory bus. Note that column-wise transfer can fundamentally improve bandwidth utilization in transfers to the cache hierarchy if the current pattern of access is column-aligned. Likewise, the total number of row-buffer operations would be reduced, further enhancing efficiencies. Thus, column-level access is fundamentally distinct from prefetching if the other columns in a fetched row of data will not be used before eviction. However, the underlying memory circuitry described in Section II only provides symmetric access to bits, not words. To support symmetric word level transfers, mats in an MDA memory are organized as "bit slices".

The logical bit split organization is shown in Fig.5. The bits of the same word are shown in the same color. The interval between the reds (bits) is a design parameter. Assuming that we are placing a red in every 8 bits, we get 64 bits of red group while transferring 64 bytes. Note that, in the conventional mode, we would also be transferring 64 bytes in the same row buffer space from the same memory array row width. Upon the request for the row, the data in the selected row is transferred to the row-buffer. Similarly, in the column mode, the bits in the column are transferred to the column-buffer. That is, in a single access, all the bits in the same word (multiple-reds in the figure) get transferred into the column buffer, as shown in Fig. 5. A detailed diagram of the bits in the words is shown in Fig. 6. To bring data from multiple bit columns into the column-buffer, we need to segment the row into *multiple groups*. The number of groups is the number of the same color bits in the same row in Fig. 5.

We propose to implement the above arrangement by adding a row selection transistor and a column selection transistor, which we call *block selectors*, as shown in Fig.6. In the column mode, the row selector transistor is turned off, and the column selector transistor is turned on. This arrangement enables the multiple column bits to be stored into the buffer in a single operation. In our implementation, we have used two additional transistors (block selectors) per 16 bits. Note that the degree of bit slicing, and associated overheads in block selectors vs. number of mats activated on a lookup are design parameters that can be freely adjusted, and the optimum interleaving can change depending on the row/column buffer capacity, the wire capacitance, and the number of block division elements area overhead. Other methods to reduce the block select elements are by rearranging words across multiple chips and using single sub array access as described in [31], [32], where a single command access actives multiple chips and multiple large arrays within chips. The power consumption can also be reduced in this approach using power-down mode as discussed in [31].

IV. Caching for MDA memories

The main focus of this work concerns the design of CPU cache hierarchies for MDA memories. We begin by providing a taxonomy of possible cache organizations that can be employed in a cache hierarchy for MDA memories, and then explain in detail the interfaces needed to support and connect each class of cache to other levels of the hierarchy.

A. Taxonomy

For the purposes of this paper, we consider a memory to be an MDA memory if it has a physical capability to supply data in multiple dimensions and we consider a cache to be a multi-dimensional MDACache if it has the capability of supporting dense accesses in more than one dimension, whether or not the physical memory arrays within the cache support multidimensional access. Thus, for caches, we will consider their "logical" and "physical" dimensionality as separate design parameters. A baseline SRAM cache in a modern processor would be considered as both logically and physically 1-D (*1P1L*). If an SRAM cache were modified to serve lines that contained column-aligned data as well as row-aligned data, both physically stored in traditional row-aligned SRAM cells, it would be considered logically 2-D, despite remaining physically 1-D (*1P2L*), and a cache built on an on-chip crosspoint could be designed to operate in a *2P2L* fashion. While a 2P1L design point is also possible, we elide discussion for brevity. For physically 2-D caches, we further subclassify them into *2P2L Dense* and 2P2L Sparse organizations, depending on whether the unit of fill is matched to the unit of allocation. Below, we discuss the cache to MDA main memory interfaces for the baseline SRAM cache and two MDACaches. Then, we also briefly discuss the additional complexities regarding connecting the cache hierarchy levels that are not from the same taxonomy class.

B. Interfaces

Below, we discuss the key interface decisions that will determine how the processor, cache levels, and MDA memory will interact.

a) Application to ISA:: We propose making the following changes at the ISA level. At the application level there are scalar, vector, and multi-dimensional variables/structures. The latter two of these will have access alignment preferences within a given code region, and Section V describes how a compiler can extract alignment preference. At the ISA level, each memory operation, either scalar or SIMD, will have both a row and column preference variant, with the compiler generating either one or the other instruction. Instructions corresponding to accesses without discerned preference will be marked as having row preference.

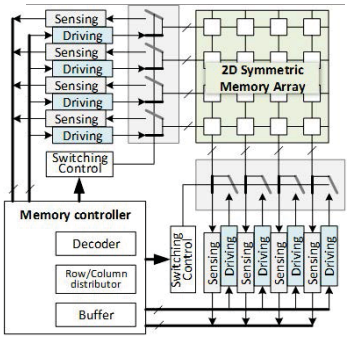


Fig. 3. A high level view of our 2D memory.

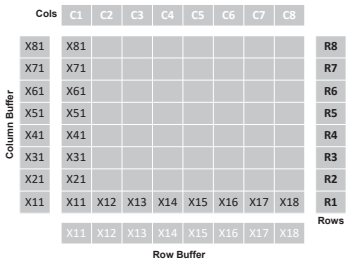


Fig. 4. 2D Memory Row / Column access.

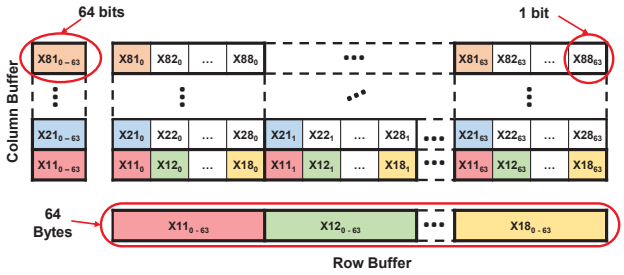


Fig. 5. 2D memory word organization.

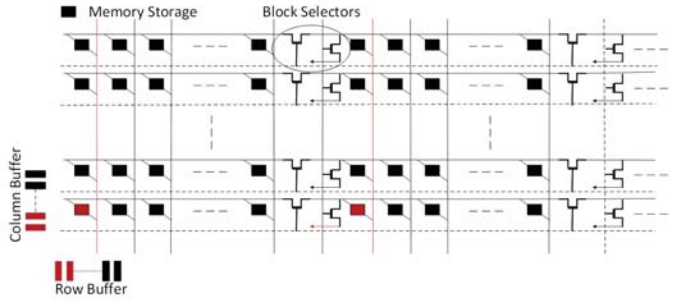


Fig. 6. 2d Memory bit-level to word-level organization.

b) Processor <–> cache hierarchy:: In a 1P2L cache, each cache line will require an additional bit of metadata to indicate whether it is "row" or "column" aligned. For scalar memory instructions, we define a hit in the cache to be based on the presence of the word within the cache line, ignoring alignment; alignment will only be used for scalar memory operations in the case of a miss by triggering a cache fill request along the preferred alignment. For vector memory accesses, the correctly aligned block must be in the cache in order for a hit to occur. In a 1P2L cache, distinct from a 1P1L cache, there exists the possibility of "data duplication" where intersecting row and column lines are co-present in the same cache level. Handling of duplicate data in 1P2L designs is a policy decision.

In a 2P2L cache, the fundamental unit of allocation is a cache line by cache line 2-D block (e.g., 512 bytes for 8 rows by 8 columns by 8 bytes per word, assuming an 8-word cache line). There is *no* data duplication, and no orientation bits need to be kept as metadata (total metadata is also reduced due to there being substantially fewer blocks than in a 1P2L cache). Section VII introduces a sparse 2-D fetch optimization, analogous to footprint cache [33] optimizations, which requires a presence bit for each row or column within a tile – the overall metadata overhead for these bits is equal to the sum of the valid and orientation bits for a 1P2L cache.

In addition, throughout the cache hierarchy, and for all the design variants tested in this work, transactions that have overlapping words should be ordered, even if the access directions are different. There are several ways to achieve this. In this work, we consider 2-D MSHRs. The access requests come out of the cache's MSHR in order, and any overlapping writes are blocked in the MSHR until the previous overlapping accesses have finished. It is also possible to maintain order at the memory without blocking the MSHR requests.

c) Intra-cache transfers:: As mentioned above, alignment preference is only considered on a cache miss for scalar memory operations. Similar to the processor-to-cache interface, a lower level cache may contain the requested data word, but in a cache line with the non-requested alignment. In contrast to the processor interface, we consider this a policy decision as to whether this constitutes a hit or miss in the lower level cache. If the higher level cache that is requesting a fill is a 1P2L cache and the lower level cache is a dense 2P2L cache and contains the requested word, it will always be able to serve a line in the orientation requested by the upper level cache. Sparse 2P2L and 1P2L upper level caches may experience partial hits due to intersecting lines in the non-preferred fill direction despite not having preferred orientation line present in the cache.

d) Cache <–> MDA memory:: We presume an MDA memory interface that will perform *bidirectional* transfers organized along cache-line sized row or column data chunks. 1P2L caches requesting an oriented fill from MDA main memory will always receive the line in the requested orientation. Similarly, writebacks can be performed in either row or column alignment, due to MDA Memory symmetry. The memory controller will track the timing constraints in both the row and column buffers. The MDA memory will respond using critical-word first transfer.

2P2L blocks will be transfered in either a row or column sequence, critical word, critical row/column first. In dense 2P2L fill, all rows/columns within the 2-D block will follow after the one generating the initial miss. In sparse 2P2L, data will only be transferred by demand or prefetch into the allocated 2-D block. Dirty/clean tracking for both dense and sparse 2P2L is done at 2-D block level, but sparse 2P2L can elide writeback of data has never been filled.

C. Selected Design Points

The caches described in Section IV can be used to form several different cache hierarchies in combination with a 2-D MDA main memory, even when restricting the number of cache layers to two. Below, we select specific instantiations that will help us to begin explore the broader design space. The combinations of interest include:

- **Design 0 – 1P1L L1 with 1P1L LLC (Baseline)**

An MDA memory can also support single dimensional access

without significant overheads. Note that the preferred memory layouts for a 1P1L software and cache optimized set of access patterns and a *P2L software and cache optimized set of access patterns will be distinct. Our experiments indicate that running a 1P1L cache hierarchy with a *P2L optimized memory could incur average slowdowns on the order of 2x, due to the mismatch between data layout and access pattern as well as extra data traffic caused by padding (see Section V). For the scope of this paper, *we will always use the memory layout optimized for the appropriate logical dimensionality of the cache hierarchy.*

• Design 1 – 1P2L L1 with 1P2L LLC

A uniformly 1P2L cache hierarchy represents the simplest translation of multidimensional access onto a modern SRAM cache technology. Metadata for each line is extended with an orientation bit to indicate either a row (unit) or column stride among consecutive words in the cache line. Key challenges include: duplicate items in cache, tag checking complexity, and the additional metadata requirement.

For physically 1D, logically 2D caches, both row and column lines are stored in the cache as dense sequences of words. To differentiate between the different access/data directions (and effective stride), an additional bit per cache line is employed to mark each cache line's direction status, as shown in Fig. 7, which is checked along with the tag on access.

Division of the physical address into tag and set mapping also has to be reconsidered to account for a mix of potentially overlapping row and column data. It is possible to utilize a common set of index bits for a simultaneous row/column lookup (*Same-Set* mapping), but doing so maps all rows and columns in a 2-D block into the same set, making it impractical for lower associativity caches. Moreover, doing so would potentially increase the complexity of cache hit logic, as up to two (read) hits in the same set would have to be correctly handled. Alternatively, rows/columns of a 2-D block can be mapped into different sets (*Different-Set* mapping), while the tag is kept the same as shown in Fig. 8. In order to avoid overheads on the hit path when using a Different-Set mapping, we generate both the row and column index in parallel and use the orientation preference bit associated with the instruction to control a 2:1 tri-state selection between them to determine which orientation is checked first (the minimal additional tri-state latency is compensated for by driver sizing). If it is a read hit in the preferred orientation, it can immediately return. If it is a miss, then the other orientation will be checked, incurring additional cycles of latency, before invoking miss handling. Even if both row and column index happen to be the same, the check is still sequential with the preferred orientation checked first. On writes, even if there is a hit in the preferred orientation, both orientations must be checked, requiring two sequential tag lookups or an additional tag port. In our design, we choose the former, as writes are not on the latency critical path. In the case of a hit in both orientations, the not preferred orientation cache line has to be either evicted or updated as well, depending on the policy. While, in this work, we consider only static mappings of orientation to instructions, the same lookup scheme would be compatible with a dynamically predicted orientation preference with no additional overheads on the cache hit path.

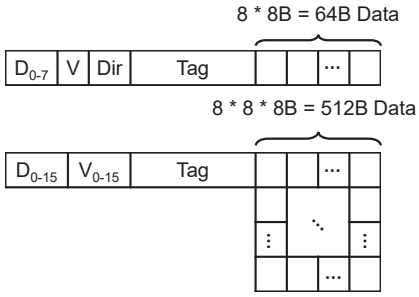


Fig. 7. Illustration of cache organization with status control bits Dirty(D), Valid(V), Access Direction(Dir). Top: physically 1D, logically 2D; Bottom: physically and logically 2D.

A new duplication challenge also exists in the 1P2L cache. Any given word could map to two intersecting cache lines of different access orientation, which means that multiple copies of the same word could be co-present in this cache. There are two significant problems that arise from this duplication: 1) how to handle writes when multiple copies exist, and 2) how to handle accesses that bring duplicate copies to the cache.

To make sure that the multiple copies stay coherent with each other, changes to the cache policy are also required. An obvious solution is modifying all copies at the same time when a change occurs and updating the new copies as they are brought to the cache, but that induces extra writes for each duplication. By implementing a writeback based method, extra writes on each modification to duplicate words are avoided. Fig. 9 shows the implemented cache policy, where the access to duplicate words are taken into account.

In our policy, duplication is allowed as long as all copies are clean. To solve the first problem, all duplicate copies except for the one accessed by the write are evicted so that modification only happens to the sole copy in the cache (shown by the transition from "Clean" to "Invalid" triggered by "Write to duplicate", indicating an incoming access is a write to a duplicate copy of this word). To solve the second problem, any modification to the sole existing copy is propagated back (write back) to the lower level cache or memory before bringing in the duplicate copy with the updated value from the lower level (the transition from "Modified" to "Invalid" triggered by "Write to duplicate", and the transition from "Modified" to "Clean" triggered by "Read to duplicate"). The ordering is ensured by forcing an ordered access for requests with overlapping word to the next level cache or memory. This ordering is enforced by logically 2-D aware MSHRs and write buffer. Correctness is obtained by ensuring that modifications can only happen when there is only one copy of the word in the cache (if any other existed, they are evicted before the modification), and all modifications (if any) are propagated back before bringing in other copies.

In addition, to mitigate the impact of extra writebacks caused by false sharing of intersecting cache lines, 1 extra dirty bit is added for each word in the cache line. For a 64-

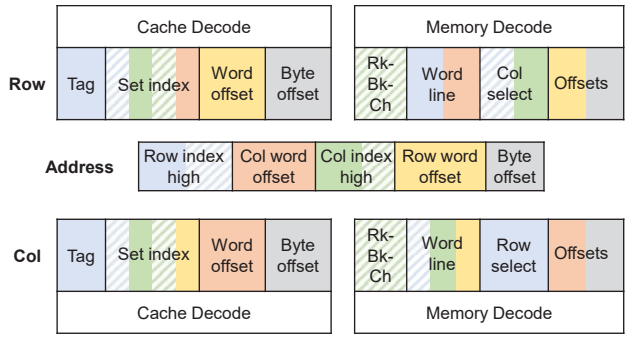
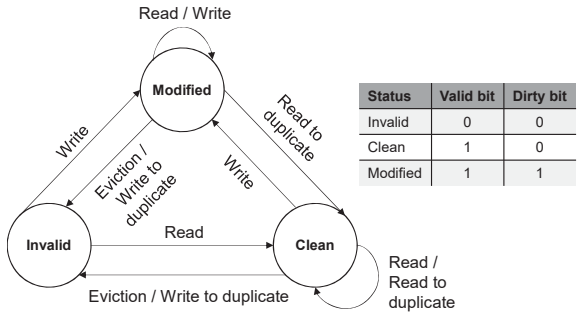


Fig. 8. Illustration of address decode mapping for row and column access directions in cache and memory, with MSB on the left. The address bits are naturally divided into row and column sections according to underlying 2D dimension.



Status	Valid bit	Dirty bit
Invalid	0	0
Clean	1	0
Modified	1	1

Fig. 9. Write back based cache policy with accesses to duplicate words taken into account. The table shows the meaning of different cache status, arrows represents status transitions, with texts representing the actions that trigger the transitions.

byte cache line with 64-bit words, the overhead is 8 bits per 64 bytes.

• **Design 2 – 1P2L L1 with 2P2L LLC**

Utilizing a physically 2D cache as the LLC simplifies some considerations regarding data duplication and tag checking complexity for the utilization-centric lower level caches. Key challenges include large unit transfer cost and increased conflict sensitivity due to reduced number of cache frames.

Allocation for a 2P2L cache works very much like a normal cache with a larger cache line size. Assuming that the 1P2L upper level cache line size is 64 bytes, then to ensure the 2D alignment of physically 2D cache on a cache line size level, the physically 2D cache's cache line size becomes 512 bytes (8 rows x 8 columns x 8 bytes/word). These 512 bytes can store a tile of aligned data with 64 bytes accessible in either a row or column direction. Compared to physically 1D, logically 2D cache, the problems of duplication and write/miss induced eviction can be eliminated.

However, the 512 byte allocation and transfer size can pose some challenges. Consider the scenario where the upper level 1P2L cache writes back a cache line, and the lower level 2P2L does not hold the corresponding 2-D block in which this line resides. Bringing in the rest of the large cache line from memory is costly, while writing through directly to the memory loses writeback and caching opportunities. Allowing sparse fill in the 2-D block mitigates these problems. On-demand or predictive prefetch methods for filling in 2-D blocks can reduce total data transfer costs, while those regions of memory that exhibit both column and row preferences in temporal proximity can still reap benefits from the subset of 2-D blocks that have been densely filled to support mixed-direction accesses.

To keep track of the status of the partial small cache lines inside a large cache line, additional status bits are added for each small cache line of each access direction, as shown in Fig. 7. With row and column bidirectional access, 16 valid bits are used to keep track of the content of a large cache line, which converts to an overhead of 16 bits per 512 bytes. Additional dirty bits can also be added to save write back bandwidth. Note that, similar to the physically 2D main memory design, We will need to add extra block selector transistors as described in Fig. 6 to enable the column transfer to the 1P2L upper level cache.

• **Design 3 – 2P2L L1 with 2P2L LLC**

While this design point would eliminate all complications concerning duplicated data within a cache level and shift to a fundamentally 2-D block-oriented allocation/transfer model, it would require utilizing a crosspoint memory for the L1 caches. As it is unclear, at this time, that doing so represents a practical design for mature NVM technologies, we leave the consideration of this design point to future work.

V. Software Support

In this section, we explain the compiler support necessary to take full advantage of an MDA-aware cache hierarchy. We can summarize our compiler support under three main items:

• **Access Direction Prediction:** To convey the row and column access information to the architecture, the compiler needs to analyze the data access patterns and extract the direction of memory accesses for each data structure. When focusing on frequently-used computational kernels, detecting the access pattern direction boils down to determining the set of subscript positions (for an array) in which the index of the innermost loop appears. For instance, let us assume that an array X is stored in memory in a row-major layout (as in C-language) and an array access $X[i][j]$ is enclosed by a loop nest where j is the innermost index. Since j appears only in the second subscript (which corresponds to the *fastest-changing* dimension in a row-major memory layout), the compiler can deduce that this access is a *row-wise* access, that is, for a given row (i) of this array structure, the innermost loop (j) traverses the elements in that row. In comparison, for the same loop and array structure, access patterns such as $Y[j][i]$ or $Z[i+j][i+2]$ can be identified by the compiler as *column-wise* accesses. Once the access directions are extracted by the compiler, they are passed to the hardware by setting a bit ("0" indicating column and "1" indicating row access) in the corresponding load/store operations.

• **MDA-memory Compliant Memory Layout:** Another type of compiler support is needed to match the dimension sizes of the array data structures to the dimensions of the MDA memory. This is to *align* the data in memory space such that two data elements, for example, $X[i][j]$ and $X[i+1][j]$ that map to the *same column* (i.e., the jth column) but consecutive

rows in the array structure need also map to the *same column* in the MDA memory structure. The compiler transformation we employ for this purpose is *array padding* [34], originally developed to reduce conflict misses in low-associative caches. In particular, in this work, we use intra-array padding, which basically involves (say, for a 2-D array) adding extra elements to each row until array references that denote the same column in the array space (such as those given above) map to the same column in our MDA memory structure. While, traditionally, such data layout optimization techniques have been used in the past to improve cache performance by reducing conflict misses [1]–[3], [5], [6], in this work, we use it for an entirely different purpose, as explained above.

•**Vectorization:** The last compiler optimization we employ is vectorization [35]. Current architectures implement vectorization in their SIMD units. In this work, we use vectorization as is, except that since our architecture allows column-wise reads in one shot, we apply vectorization in the column direction as well in the row direction. In contrast, in state-of-the-art compilers, vectorization is not usually used in column-wise accesses, as doing so typically involves first copying multiple elements (from different rows, in the same column) to consecutive locations, which may be too costly and can easily offset the potential benefits of vectorization. In other words, our approach expands vectorization opportunities that can be targeted by the compiler.

A. Applying the proposed approach

To understand how and when our proposed approach will be applied, consider the following example involving matrix multiplication. Note that, for expository simplicity, a naive MxM algorithm is used.

```
1:  function MATRIX_MULTIPLICATION(MatR, MatC, N)
                    ▷ Input matrices MatR, MatC of size N by N
                    ▷ MatR is accessed in rows, MatC in columns
2:      Let MatOut be a new matrix of size N by N
3:      for i ← 0, N − 1 do
4:          for j ← 0, N − 1 do
5:              sum ← 0
6:              for k ← 0, N − 1 do
7:                  sum ← sum + MatR_{i,k} ∗ MatC_{k,j}
8:              MatOut_{i,j} ← sum
9:      return MatOut
10: end function
11: function MDA_EXAMPLE(N)
12:     Let A, B, C, D, E be matrices of size N by N
13:     ...
14:     C ← MATRIX_MULTIPLICATION(A, B, N)
                    ▷ Row accesses direction for A, column for B
15:     ...
16:     E ← MATRIX_MULTIPLICATION(B, D, N)
                                ▷ Row for B, column for D
17:     ...
18: end function
```

While memory operations in the MatrixMultiplication function have strong row/column preference, the data structures this function will be called on **may not**, as in the case of matrix B. As matrix B lacks a globally dominant access preference, memory layout in column-major form could require the insertion of a transpose operation between the two calls to the matrix multiplication routine or the generation of multiple versions of the MatrixMultiplication function with asymmetric performance expectations. In our approach, however, all matrices A–E can be laid out in row-major form, provided that logical column and memory layout of column alignments are preserved. Through static analysis, the instructions accessing $MatC_{k,j}$ are identified and annotated with column access preference, and all other instructions retain row access preference.

The differences between column and row access preference for $MatC_{k,j}$ can be substantial, and are not limited to cache hit rate improvements. Specifically, being able to perform transfers of column data between layers of the memory hierarchy can better utilize bandwidth both among cache levels and between the cache hierarchy and memory by only bringing in the data words in the current direction of access locality. Further, by consolidating column data, vector operations can be performed on column-aligned data as readily as on row-aligned data. Assume, for instance, that each cache line can hold 8 elements of matrix $MatR$ or $MatC$, and that the cache is empty in the beginning. For $k \leftarrow 0, 7$, the accesses to $MatR_{i,k}$ can be satisfied with 1 access to the memory that brings back 1 cache line into the cache. However, 8 row accesses are needed to get $MatC_{k,j}$, each bringing back 1 cache line of $MatC_{k,m}$ to $MatC_{k,m+7}$, where $m = \lfloor j/8 \rfloor$. With column access enabled, a single access can get all 8 element of $MatC_{k,j}$, reducing number of data transfer to $1/8$. Furthermore, the size that the data occupies in the cache is also reduced to $1/8$, increasing the chance of useful data remaining in the cache and having higher cache hit rates.

In addition to the benefits seen in this example, there are other scenarios where decoupling layout and access direction preferences could be particularly useful. One such example is in column-IO database [36] layouts, wherein the (logically) column data is of a single data type, whereas a row may contain columns of several data types. As such, there are substantial compression opportunities within a column that are not present for rows. Providing similar cost accesses to both row and column access patterns would allow for greater flexibility in preserving simplicity of row-column addressing within such tables while still being able to benefit from data-type-specific column-compression.

In cases where a data reference in the target code does not exhibit a strong row or column preference that can be detected by the compiler, we can employ profiling. More specifically, profiling can be used to extract directional bias and then the corresponding static load/store instructions can be annotated (with access preference information) as suggested by the profiler.

VI. Modeling and Methodology

A. Memory/Cache Modeling

To preserve column alignment within the same bank, and preserve to a certain degree the bank, rank, and channel level

TABLE I
EXPERIMENTAL SETUP

Gem5 configurations	
CPU	X86 architecture, OoO, 3 GHz
L1 D-/I- cache	32KB, 4 way associative 2-cycle tag lookup , 2-cycle data access Parallel tag/data access
L2 cache (256KB)	8 way associative 6-cycle tag lookup, 9-cycle data access Sequential tag/data access
L2 (2MB) L3 (1/1.5/2/4MB)	8 way associative 8-cycle tag lookup, 12-cycle data access Sequential tag/data access
Main memory	4GB, NVMain simulator
Simulation mode	Syscall Emulation
NVMain configurations	
Memory controller	FRFCFS-WQF
Device Config	STT-RAM
Memory Size	1GB/channel x 4 channels
Row buffer policy	Open page

parallelism, we modify the main memory decode as shown in Fig. 8. In the address, the "Byte offsets" signifies the byte in a 64-bit word, and the 3-bit "Row word offset" and "Col word offset" indicates the word in a 64-byte row and column cache line respectively. Together, they define a tile of 8 cache lines where we have physically continuous column cache lines as well as row cache lines. By not using "Row word offset" or "Col word offset" in the mapping to bank, rank or channel, it is ensured that any interleaving method employed will interleave on a tile basis, and will not disturb the column alignment within a column cache line. We push the selection of bank, rank, and channel bits as much as possible toward the LSB to enhance channel, rank and bank-level parallelism. The remaining bits are partitioned between "Offsets" for bytes in a cache line, "word line" for row/column selection, and finally we have the "Row select" and "Col select" that selects which cache line in a physical column and row respectively. The interleaving, as shown in Fig. 8, can be denoted as R:C:BK:RK:CH where a column aligned tile is the unit of interleaving, instead of a row-wise cache line. To use the bidirectional feature of the main memory, we pass an identifier to the main memory indicating that a fetch is for a column or a row. The memory controller will ensure that the column is physically aligned as a column.

We made the following modifications in cache modeling. In addition to the extra bits and policies described in Section IV-C, an extra delay has been added to the 1P2L different set mapping cache implementation to account for the extra tag accesses. In the case of a scalar cache access, if it misses in the desired direction, an additional tag access and latency is added to see if there is a "hit" in the other direction. In the case of a SIMD access, if it misses (in the desired direction), then additional tag accesses and latency are added to check for the existence of any dirty intersecting cache lines of the other orientation that may need to be written back. As an example, for a 1P2L cache, the number of additional tag accesses is the number of words in a cache line (8 in the evaluation). For a write, additional tag accesses and latency are added for potential eviction; the tag check overheads are the same as a read miss. Note that all the extra latencies are incurred either on miss or write, which are off the critical path. Moreover, as will be seen in the next section, duplication is very rare for the examined workloads, which limits the observed overheads of even the above heavyweight eviction approach.

B. Experimental Setup

We evaluated our proposed MDACache policies using a set of benchmarks featuring *both* row and column access affinities. Our processor and cache simulation infrastructure is based on GEM5 [37], while the MDA main memory is modeled using NVMain [38]. Table I summarizes the various simulation parameters used in our evaluations. The evaluated benchmarks include: *sgemm, ssyr2k, ssyrk, strmm, sobel, htap1, htap2*. While sgemm, syyr2k, syyrk and strmm belong to LAPACK BLAS [39], the sobel benchmark evaluated is a basic Sobel filter for vertical traversal. The htap1 and htap2 are analytical and transactional processing benchmarks from [40]. All input matrices (except htap) are 256 x 256 x 64-bit/ 512 x 512 x 64-bit; Htap uses 2048 x 256 x 64-bit/ 2048 x 512 x 64-bit.

We use NVMain [38] to model an STT crosspoint array for our MDA main memory. Prior work [16] has shown crosspoint memory implementations using STTRAM arrays and bidirectional selectors [7], [41] suitable for MDA memories. Note that we expect similar improvements with other crosspoint based technologies (e.g., ReRAM, PCM) as well. We accounted for the column decoder delay by adding an additional cycle to address translation.

VII. RESULTS

In this section, we present a detailed experimental evaluation of the proposed cache management schemes using a set of widely used computation kernels as our benchmarks. Results presented in this section are performed on a system with 3-levels of caches, with 32KB L1, 256KB L2 and 1MB L3, using 512x512 input set, unless stated otherwise. To illustrate that the benefit from the proposed scheme outweighs the benefit of prefetching, we evaluate our proposed 1P2L and 2P2L designs without prefetching, whereas the baseline 1P1L cache hierarchy is evaluated with prefetching enabled. We expect that prefetching optimizations for MDA memories/caches are likely to constitute an area of research unto themselves and their exploration is beyond the scope of this paper. **In all the evaluation results presented below, the L1 data cache is always physically 1-D (SRAM)**, with logical dimension 1 in the baseline and 2 in all MDACache hierarchies, the same is true for L2 caches that are not LLC. 1P1L and 1P2L LLCs are SRAM, and 2P2L LLCs modeled with STT parameters. All logically 1-D experiments use a 1-D optimized memory layout and all logically 2-D experiments use a 2-D optimized memory layout.

•**1P2L:** The graph in Fig. 12 plots the execution cycles of our benchmarks, *normalized* with respect to the conventional (1P1L) cache hierarchy with data prefetching. We use 1P2L to denote different set mapping cache modeling, and 1P2L_SameSet for same set. It can be observed from this graph that 1P2L outperforms 1P1L in all benchmark

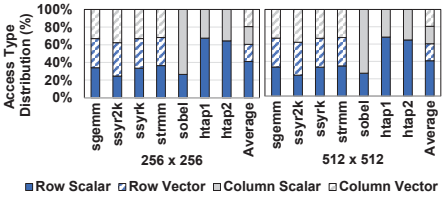


Fig. 10. Access orientation and size preferences in the target workloads, by data volume. Accesses are categorized by row and column preferences for both scalar and potential vector accesses.

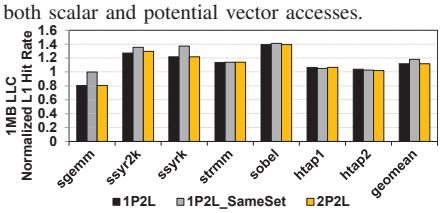


Fig. 11. L1 hit rates normalized to 1P1L (with prefetching) with 1MB LLC, (512x512) input set.

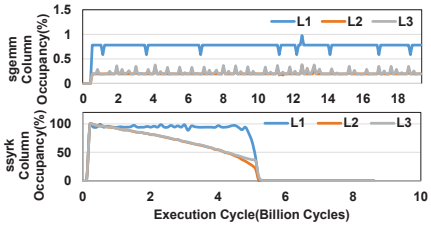


Fig. 15. Col. vs. row cache occupancy over time

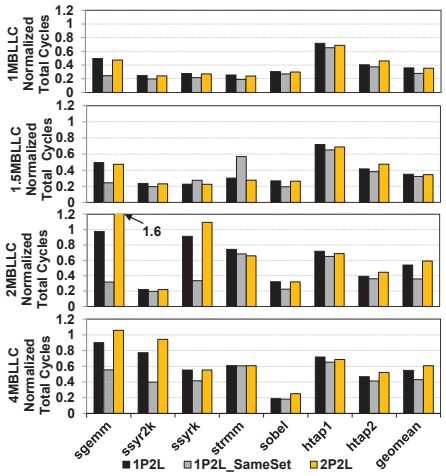


Fig. 12. Latency normalized to 1P1L (with prefetching) with 1MB, 1.5MB, 2MB, 4MB L3s, all with 256K L2 and 32K L1 for non-cache-resident (512x512) input set.

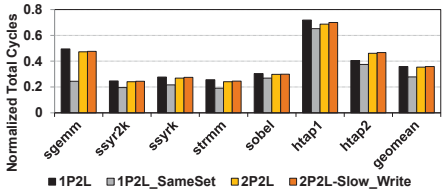


Fig. 16. Impact of highly-asymmetric write latency for 2P2L

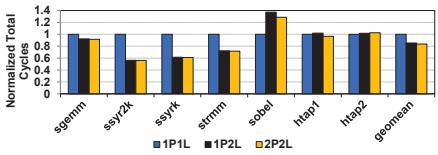


Fig. 13. Normalized latency for 2MB L2(LLC) and cache resident (256x256) input sets

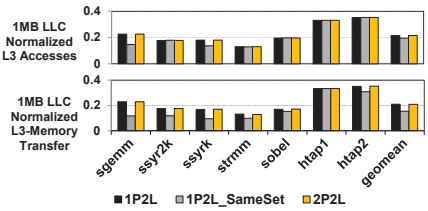


Fig. 14. L3 accesses and L3-memory transfer normalized to 1P1L (with prefetching) with 1MB LLC, (512x512) input set.

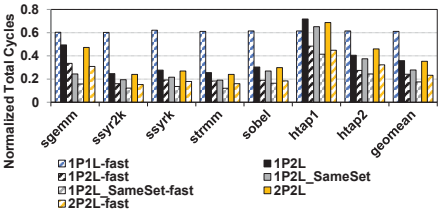


Fig. 17. Benefits compared with, and in the presence of, improved main memory performance.

programs tested, and at worst reduces the execution time by 3% of baseline with 2MB LLC. With 1/1.5/2/4MB LLC, the execution time is reduced by 64/65/46/45% on average for 1P2L (72/68/64/57% for 1P2L_SameSet). Moreover, the 1P1L baseline has prefetching enabled, hence the benefits of logically 2-D access extend beyond mere prefetching for column stride data. Note also that, a prefetcher with high coverage, even though it can eliminate a large number of misses – just as our 1P2L – it still needs to issue multiple requests to bring a column data, thereby increasing the traffic between cache and memory.

To lend insight to the source of these improvements, we show, in Fig. 10, the breakdown of access preferences, for each benchmark, into four categories covering each combination of preference row, column and size scalar, vector (see Sections V and IV for the discussion of how these preferences are conveyed from software to hardware and how they are considered by hardware). The most critical observation from these results is that the 1P2L cache exercises column preference (which is not an option in the 1P1L cache) in *all* benchmarks. In fact, on average, we see that column preferences constitute about 40% of total data accesses between both scalar and vector forms.

While the results presented in Fig. 10 indicate that, when running on a 1P2L cache, our benchmarks exercise column preference, it is also important to understand what this really means in terms of the cache behavior. Fig. 11 and Fig. 14 present the L1 hit rate and total number of L3 accesses with a 1MB LLC, respectively. While 1P2L does not guarantee a better L1 hit rate than 1P1L for all benchmarks, it is 12% (18% for 1P2L_SameSet) better, on average, and many misses to the same column are combined into one column access in the MSHR, resulting in substantially fewer L3 accesses (only 22% of 1P1L, only 20% with 1P2L_SameSet, on average). As shown in Fig. 14, this traffic reduction is also inherited to the L3 to memory transfer, with the total bytes of memory transfer for 1P2L reduced to only 21% of 1P1L (15% for 1P2L_SameSet).

Different benchmarks exercise column preference differently and in a time-varying fashion. To illustrate this point, Fig. 15 plots the utilization of column-cached lines over time (x-axis) in two of our benchmarks, namely, sgemm and ssyrk (with 32KB L1, 256KB L2 and 1MB L3 LLC system). We see that these two benchmarks exhibit distinct patterns; in sgemm, the column preference is stable over the execution period, whereas in ssyrk, it first increases and then decreases (due to neighboring loop nests exhibiting different preferences in the later part of the execution). Moreover, despite sgemm having a substantial column preference (Fig. 10), the loop ordering it has means that only a few of those columns are present in the cache at a time, while row-oriented data cycles through.

We can conclude from this analysis that improvements in both hit rates (Fig. 11) and data traffic (Fig. 14) eventually translate to significant savings in execution cycles (Fig. 12).

•**2P2L:** Recall from Section IV that our 2P2L implementation allocates and evicts at 2-D block level, but each block is only filled one row or column at a time based on demand. The

results from Fig. 12 indicate that, the worst performance is 1.6 times than the base line. Note that 2MB is the local working set size, and we suspect the variances in 2MB L3 results are due to this edge case.Also we observe equivalent performance as baseline when we increase the L3 to 4MB for the same benchmark. On average, with 1/1.5/2/4MB LLC, the execution time is reduced by 65/66/41/39%. The 2P2L generates slightly better results than the 1P2L different set cache on multiple benchmarks across most of the LLC variations. Compared to the 1P2L implementation, 2P2L has the advantage of having less miss overhead, but higher set conflicts.

VIII. Sensitivity Experiments

We explore the sensitivity of the proposed scheme along three dimensions. First, we present the sensitivity of results to working-set size / LLC size relationships, thereby gaining insight into the degree to which there are bandwidth benefits from not only memory to cache transfers, but also cache to cache transfers. We then show the sensitivity of 2P2L results to on-chip NVM read/write asymmetry, taking into account the larger write latency of many on-chip NVM technologies. Finally, we evaluate sensitivity to main memory read/write speed to explore both future scalability and gain insight into the viability of the proposed approach if MDA-capable memory technologies remain slower than other alternatives. Note sensitivity study results uses 32KB L1, 256KB L2,1MB L3 LLC system, 512x512 input set unless stated otherwise.

Working set size: In addition to the varying degrees of cache non-residency (512x512 input matrix size with 1/1.5/2/4MB LLCs) configurations explored in Figure 12, we also consider the sensitivity of our approach to an entirely cache-resident working set. Fig. 13 shows our approach evaluated for a smaller 256x256 input, with a 2MB L2 as the LLC. Latency is still reduced, on average. The 1P2L performs better than the 1P1L base line with execution time reduced by 14% on average, and 2P2L performs slightly better with execution time reduced by 16% on average. Compared to the non-resident case, cache resident workloads take limited advantage of the L2 to memory bandwidth reduction, which explains the smaller improvement. However, the bandwidth reduction between L1 and L2 is preserved, providing better improvements to some benchmarks.

On-chip NVM read/write asymmetry: The proposed 2P2L scheme is not dependent on a specific NVM technology, and possible on-chip NVM technologies are known to exhibit a wide range of write/read latency ratios [25]. Results presented in section VII assume symmetrical write and read latency for the 2P2L on-chip cache. Fig. 16 compares the symmetric 2P2L with one where writes takes 20 additional cycles equivalent time [42] compared to reads, using a 256KB L2/ 1MB L3 cache (cache non-resident). 2P2L with asymmetric write latency performs slightly worse than symmetric 2P2L, with a difference of 0.4% on average. While a longer write latency does not appear to change the trend between the baseline and 2P2L implementations, 2P2L may not be favorable for technologies with very high asymmetry ratio.

Main memory speed: Among the benefits of the proposed schemes is a reduction of transfers between cache and memory. The extent of this benefit is affected by the main memory speed, and Fig. 17 shows how a 1.6× faster main memory would influence the results. This experiment allows us to evaluate whether our approach could still be beneficial if there was a substantial memory performance gap between MDA and non-MDA memories, and help predict how it may scale with memory future technologies. This evaluation uses a 256KB L2/1MB L3 cache (cache non-resident). Similar benefit trends are present with a faster memory, with 1P2L-fast reducing 61% of the execution time over 1P1L-fast. Moreover, 1P2L, even with the baseline memory, outperforms 1P1L-fast by reducing 41% of the execution time, indicating that the approach is promising even if MDA memories remain slower than comparable memory alternatives.

IX. Related Work

Since memory technologies and organizations that enable MDA memories/caches have already been discussed in Sections II and III, we do not repeat them here. Instead, below, we discuss the other prior efforts related to this work in two categories: bandwidth reduction techniques, and row-buffer optimizations.

A. Bandwidth Reduction Techniques

Data Layout Optimizations: There have been many prior works which optimized for the memory bandwidth by changing the data layouts. At a high-level, all these works proposed augmenting data layouts using either row or column stores [43]. Seshadri et al [40] identified that, though the row/column store ensures the data accessed together are physically co-located, the traditional DRAM interfaces for these non-unit stride workloads still cause ”non-useful” data to be fetched from DRAM, thereby wasting the memory bandwidth. GS-DRAM optimized such bandwidth wastage by proposing an advanced scatter/gather operation support in the DRAM substrate. Though their mechanism saves the memory bandwidth, it also requires significant changes to the DRAM substrate to enable the scatter/gather operations unlike our MDA memory where the column accesses are inherently enabled by the underlying 2D memory technology.

Sparse Fetch Optimizations: Another relevant area of work which our proposed solutions can benefit from are sparse cache line fetch optimizations like footprint [33] and dense footprint [44] caches. These proposals identify the useful cache lines within a page to reduce the amount of non-useful cache lines fetched from memory to be stored into stacked DRAM. We can envision that our 2D memory-based tile fetch can easily be enhanced with ideas from the footprint and dense footprint [33], [44] caches to further alleviate the memory bandwidth consumption.

Prefetching: The column fetch mode enabled by our MDA memories will reduce to a stride-prefetcher [45]–[47] in a conventional 1 dimensional memory with data laid out appropriately. However, the stride in a column access would be

same as the size of a page, for example 4KB, and the prefetch degree would match the cacheline size across different pages. Depending on the column strides exhibited by the workload, the stride can be multiples of the page size (e.g., 4KB, 8KB, etc).

Other techniques: Memory/Cache Compression techniques are another promising direction to enhance overall memory and cache bandwidth. Various compression techniques have been proposed by researchers [48]–[50], which evaluate the trade-offs in terms of compression-ratio vs decompression latency. Though our proposed MDA cache and memory naturally fetch contiguous row, column or tiled data, these compression techniques can improve the overall bandwidth and can increase the effectiveness of our MDA proposal and hence are complementary. Texture caches [51], [52], like 2P2L, exploit multi-dimensional tiled locality, but are generally read-only, reduce bandwidth but have high latency (L1 TC hit times of more than 200 cycles reported in [52]) and are not physically multi-dimensional.

B. Row buffer Optimizations

Multiple sub-row buffers: Gulur et al [53] proposed a technique to split a row buffer into four multiple sub-row buffers to increase the row buffer hitrate. Though their mechanism can reduce the overall memory access latency by increasing the row buffer hit rate for some workloads, for applications which perform strided column accesses, their mechanism cannot provide the desirable row-buffer coverage. This is because the DRAM substrate employed in [53] still warrant multiple rows to be activated unlike in our MDA Memory where the entire column can be loaded in to the column buffer with one operation. We implemented a multiple row-buffer scheme and found it to have a less than 1% impact; while such schemes are very useful for multiprogrammed workloads, single-application, single thread scenarios are less sensitive. An investigation of our techniques on parallel workloads would examine these approaches in greater detail.

Micro-Pages: Sudan et al in [54] proposed a hardware-based remapping technique where the most frequently accessed cachelines are coalesced in to one page. Such a coalesced page with multiple frequently accessed cache lines aims to maximize row-buffer hit rates. Their mechanism employs a hardware remapping structure which contains the meta-data that remaps the original page address to the newly coalesced page address. Though their approach is orthogonal to MDA memories, such mechanisms can be complement our 2D memories to enhance their overall effectiveness.

X. Conclusions and Future Work

Emerging crosspoint technologies offer new opportunities, not only for constructing dense memories, but also in supporting multidimensional access. However, capitalizing on this opportunity requires changes in both compiler support for new memory layouts and caches capable of connecting application access alignment preferences to MDA memories. In this paper, we presented a pair of logically 2-D cache designs, one physically 1-D and easily realized with SRAM, and the other leveraging an on-chip MDA memory. We examined the benefits of 1P2L caching, and showed that there are substantial improvements possible through supporting multidimensional access, and that these benefits are amplified by enabling existing vectorization approaches to vectorize along multiple dimensions. Our results indicate that logically 2-D caching using physically 1-D SRAM structures combined with vectorization in both row and column directions provides 72% average reduction in execution time over a traditional cache system interfacing with an MDA memory. We then show that utilizing an MDA memory technology on-chip to implement a cache that is both logically and physically 2-D, using sparse block fetch, can provide a 65% reduction.

One interesting direction for future work would be enhancing the performance of 2P2L caching via iteration space tiling [55], a compiler optimization that improves cache performance by dividing the iteration space of a given loop nest into smaller blocks (tiles) and reusing the data elements accessed by each tile. In our case, the compiler can tile a loop nest such that the tile size (in each dimension) matches the 2-D block size used by the 2P2L cache or a desirable multiple thereof. We expect such hardware-software collaborative tiling to generate better results than software tiling or hardware tiling (2P2L) alone.

References

[1] E. Z. Zhang, H. Li, and X. Shen, "A study towards optimal data layout for gpu computing," in *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2247684.2247699

[2] M. T. Kandemir, A. N. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee, "Enhancing spatial locality via data layout optimizations," in *Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646663.700129

[3] S. Rubin, R. Bodík, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," *SIGPLAN Not.*, vol. 37, Jan. 2002. [Online]. Available: http://doi.acm.org/10.1145/565816.503287

[4] J. Liu, J. Kotra, W. Ding, and M. Kandemir, "Network footprint reduction through data access and computation placement in noc-based manycores," in *Proceedings of the 52Nd Annual Design Automation Conference (DAC)*, 2015.

[5] D. Cho, S. Pasricha, I. Issenin, N. Dutt, Y. Paek, and S. Ko, "Compiler driven data layout optimization for regular/irregular array access patterns," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1375657.1375664

[6] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang, "A data layout optimization framework for nuca-based multicores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155677

[7] A. Aziz, N. Jao, S. Datta, and S. K. Gupta, "Analysis of functional oxide based selectors for cross-point memories," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, 2016.

[8] S. George, K. Ma, A. Aziz, X. Li, A. Khan, S. Salahuddin, M.-F. Chang, S. Datta, J. Sampson, S. Gupta, and V. Narayanan, "Nonvolatile memory design based on ferroelectric fets," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016.

[9] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016.

[10] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016.

[11] M. Imani, S. Gupta, A. Arredondo, and T. Rosing, "Efficient query processing in crossbar memory," in *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017.

[12] C. J. Lin, S. H. Kang, Y. J. Wang, K. Lee, X. Zhu, W. C. Chen, X. Li, W. N. Hsu, Y. C. Kao, M. T. Liu, W. C. Chen, Y. Lin, M. Nowak, N. Yu, and L. Tran, "45nm low power cmos logic compatible embedded stt mram utilizing a reverse-connection 1t/1mtj cell," in *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 2009.

[13] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, vol. 37, 2009.

[14] M. Zwerg, A. Baumann, R. Kuhn, M. Arnold, R. Nerlich, M. Herzog, R. Ledwa, C. Sichert, V. Rzehak, P. Thanigai, and B. O. Eversmann, "An 82μa/mhz microcontroller with embedded feram for energy-harvesting applications," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*. IEEE, 2011.

[15] T. Y. Liu, T. H. Yan, R. Scheuerlein, Y. Chen, J. K. Lee, G. Balakrishnan, G. Yee, H. Zhang, A. Yap, J. Ouyang, T. Sasaki, S. Addepalli, A. Al-Shamma, C. Y. Chen, M. Gupta, G. Hilton, S. Joshi, A. Kathuria, V. Lai, D. Masiwal, M. Matsumoto, A. Nigam, A. Pai, J. Pakhale, C. H. Siau, X. Wu, R. Yin, L. Peng, J. Y. Kang, S. Huynh, H. Wang, N. Nagel, Y. Tanaka, M. Higashitani, T. Minvielle, C. Gorla, T. Tsukamoto, T. Yamaguchi, M. Okajima, T. Okamura, S. Takase, T. Hara, H. Inoue, L. Fasoli, M. Mofidi, R. Shrivastava, and K. Quader, "A 130.7-mm 2-layer 32-gb reram memory device in 24-nm technology," *IEEE Journal of Solid-State Circuits*, vol. 49, 2014.

[16] W. Zhao, S. Chaudhuri, C. Accoto, J.-O. Klein, C. Chappert, and P. Mazoyer, "Cross-point architecture for spin-transfer torque magnetic random access memory," *IEEE Transactions on Nanotechnology*, vol. 11, 2012.

[17] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono, "An 8 mb multi-layered cross-point reram macro with 443 mb/s write throughput," *IEEE Journal of Solid-State Circuits*, vol. 48, 2013.

[18] D. Kau, S. Tang, I. V. Karpov, R. Dodge, B. Klehn, J. A. Kalb, J. Strand, A. Diaz, N. Leung, J. Wu, S. Lee, T. Langtry, K. wei Chang, C. Papagianni, J. Lee, J. Hirst, S. Erra, E. Flores, N. Righos, H. Castro, and G. Spadini, "A stackable cross point phase change memory," in *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 2009.

[19] K. Bong, S. Choi, C. Kim, S. Kang, Y. Kim, and H.-J. Yoo, "14.6 a 0.62 mw ultra-low-power convolutional-neural-network face-recognition processor and a cis integrated with always-on haar-like face detector," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*. IEEE, 2017.

[20] R. Naous, M. Al-Shedivat, E. Neftci, G. Cauwenberghs, and K. N. Salama, "Stochastic synaptic plasticity with memristor crossbar arrays," May 2016.

[21] D. Kuzum, R. G. Jeyasingh, B. Lee, and H.-S. P. Wong, "Nanoelectronic programmable synapses based on phase change materials for brain-inspired computing," *Nano letters*, vol. 12, 2011.

[22] "Intel launches optane memory m2 cache ssds for client market," https://www.anandtech.com/show/11227/intel-launches-optane-memory-m2-cache-ssds-for-client-market, accessed: 2017-11-11.

[23] H. Noguchi, K. Ikegami, N. Shimomura, T. Tetsufumi, J. Ito, and S. Fujita, "Highly reliable and low-power nonvolatile cache memory with advanced perpendicular stt-mram for high-performance cpu," in *2014 Symposium on VLSI Circuits Digest of Technical Papers*, June 2014.

[24] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das, "Re-NUCA: A practical nuca architecture for reram based last-level caches," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[25] H. Y. Cheng, J. Zhao, J. Sampson, M. J. Irwin, A. Jaleel, Y. Lu, and Y. Xie, "Lap: Loop-block aware inclusion properties for energy-efficient asymmetric last level caches," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016.

[26] "Spin-transfer torque mram technology," https://www.everspin.com/spin-transfer-torque-mram-technology, accessed: 2017-11-11.

[27] J. Song, J. Woo, A. Prakash, D. Lee, and H. Hwang, "Threshold selector with high selectivity and steep slope for cross-point memory array," *IEEE Electron Device Letters*, vol. 36, 2015.

[28] Z. Wang, D. A. Jimnez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an stt-ram-based hybrid cache," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[29] S. George, X. Li, M. J. Liao, K. Ma, S. Srinivasa, K. Mohan, A. Aziz, J. Sampson, S. K. Gupta, and V. Narayanan, "Symmetric 2-d-memory access to multidimensional data," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, June 2018.

[30] C. P. Lo, W. Z. Lin, W. Y. Lin, H. T. Lin, T. H. Yang, Y. N. Chiang, Y. C. King, C. J. Lin, Y. D. Chih, T. Y. J. Chang, M. S. Ho, and M. F. Chang, "Embedded 2mb reram macro with 2.6 ns read access time using dynamic-trip-point-mismatch sampling current-mode sense amplifier for ioe applications," in *VLSI Circuits, 2017 Symposium on*. IEEE, 2017.

[31] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010.

[32] J. B. Kotra, N. Shahidi, Z. A. Chishti, and M. T. Kandemir, "Hardware-software co-design to mitigate dram refresh overheads: A case for refresh-aware process scheduling," in *Proceedings of 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[33] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485957

[34] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," *SIGPLAN Not.*, vol. 33, May 1998. [Online]. Available: http://doi.acm.org/10.1145/277652.277661

[35] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1133981.1133997

[36] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, 2009.

[37] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[38] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, July 2015.

[39] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[40] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Gather-scatter dram: In-dram address translation to improve the spatial locality of non-unit strided accesses," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO*, 2015.

[41] S. H. Jo, T. Kumar, S. Narayanan, and H. Nazarian, "Cross-point resistive ram based on field-assisted superlinear threshold selector," *IEEE Transactions on Electron Devices*, vol. 62, 2015.

[42] S. Motaman, S. Ghosh, and N. Rathi, "Impact of process-variations in sttram and adaptive boosting for robustness," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2757144

[43] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[44] S. Shin, S. Kim, and Y. Solihin, "Dense footprint cache: Capacity-efficient die-stacked dram last level cache," in *Proceedings of the 4th Annual International Symposium on Memory Systems (MEMSYS)*, 2016.

[45] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting midway: Improving cmp performance with memory-side prefetching," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[46] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[47] "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," https://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers, accessed: 2017-11-11.

[48] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[49] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[50] J. Gaur, A. R. Alameldeen, and S. Subramoney, "Base-victim compression: An opportunistic cache compression architecture," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[51] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997. [Online]. Available: http://doi.acm.org/10.1145/264107.264152

[52] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010.

[53] N. D. Gulur, R. Manikantan, M. Mehendale, and R. Govindarajan, "Multiple sub-row buffers in dram: Unlocking performance and energy improvement opportunities," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, 2012.

[54] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[55] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *SIGPLAN Not.*, vol. 26, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/106973.106981