Software-Defined Address Mapping: A Case on 3D Memory

Jialiang Zhang
jlzhang@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Michael Swift
swift@cs.wisc.edu
University of Wisconsin-Madison
Madison, WI, USA

Jing (Jane) Li
janeli@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

ABSTRACT

3D-stacking memory such as High-Bandwidth Memory (HBM) and Hybrid Memory Cube (HMC) provides orders of magnitude more bandwidth and significantly increased channel-level parallelism (CLP) due to its new parallel memory architecture. However, it is challenging to fully exploit the abundant CLP for performance as the bandwidth utilization is highly dependent on address mapping in the memory controller. Unfortunately, CLP is very sensitive to a program's data access pattern, which is not made available to OS/hardware by existing mechanisms.

In this work, we address these challenges with software-defined address mapping (SDAM) that, for the first time, enables user program to obtain a direct control of the low-level memory hardware in a more intelligent and fine-grained manner. In particular, we develop new mechanisms that can effectively communicate a program's data access properties to the OS and hardware and to use it to control data placement in hardware. To guarantee correctness and reduce overhead in storage and performance, we extend Linux kernel and C-language memory allocators to support multiple address mappings. For advanced system optimization, we develop machine learning methods that can automatically identify access patterns of major variables in a program and cluster these with similar access patterns to reduce the overhead for SDAM. We demonstrate the benefits of our design on real system prototype, comprising (1) a RISC-V processor, near memory accelerators and HBM modules using Xilinx FPGA platform, and (2) modified Linux and glibc. Our evaluation on standard CPU benchmarks and data-intensive benchmarks (for both CPU and accelerators) demonstrates 1.41×, 1.84× speedup on CPU and 2.58× on near memory accelerators in our system with SDAM compared to a baseline system that uses a fixed address mapping.

CCS CONCEPTS

• **Software and its engineering → Main memory**; • **Computer systems organization → Reconfigurable computing**.

KEYWORDS

Software defined memory, 3D memory, Address mapping

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507774

ACM Reference Format:
Jialiang Zhang, Michael Swift, and Jing (Jane) Li. 2022. Software-Defined Address Mapping: A Case on 3D Memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507774

1 INTRODUCTION

DDR-based main memory is a critical performance bottleneck in today's data centers as memory bandwidth cannot keep pace with the explosion in data. The problem becomes more pronounced with new trends in processor technology and applications: 1) Increasing the number of general-purpose cores and accelerators integrated into a single chip raises competition for access to DRAM and demands higher bandwidth from main memory. 2) Emerging applications such as data analytics and graph processing are data intensive and increasingly becoming memory bound. As such, 3D-memory is a promising alternative to tackle these problems by providing more bandwidth. The two realizations of 3D-memory are High Bandwidth Memory (HBM) [23] and Hybrid Memory Cube (HMC) [37]. These memories exploit through-silicon via (TSV)-based stacking technology and organizes memory banks into multiple independently-operated channels, offering > 10× more channel-level parallelism (CLP) than DDR memory. Hence, they can achieve > 10× higher peak hardware bandwidth. Such CLP is expected to grow more for future-generation 3D memory devices [36].

One of the key challenge of using 3D memory is how to fully exploit the CLP offered by the new parallel architecture. To increase CLP and hence bandwidth utilization, it is important to spread concurrent memory requests across as many channels as possible. This can be controlled via the address mapping in the memory controller that transforms the flat 1D of physical addresses into an internal 3D hierarchical structure of channels, banks and rows. However, in comparison to DDR DRAM, 3D memory has many more channels but smaller row buffer size. Thus, to better exploit the CLP in 3D memory, consecutive memory access (cache-line size granularity) should be sent to distinct channels via fine-grained control of data placement. The mapping has to vary with different data access patterns to fully exploit its potential: when program strides through data, the generated memory references may cause contention on one or few memory channels. Thus, we need a mechanism that can effectively predict/capture the access patterns of different data structure in a program and leverage such knowledge to support multiple address mappings for data structures with different data access patterns.

Many existing works address the data placement issue using *hardware-only* mechanisms. These methods rely on dedicated hardware to control data placement based on a *single* address mapping [1, 17, 30, 49] and apply *one global* address mapping to *all*

physical addresses after address translation or a very large address range. To predict access patterns, they use *physical* addresses that only provide a localized view (page) of accesses and cannot capture different access patterns to different data structures in *virtual* memory. Hence, they often result in limited performance gain.

There have also been a number of *software-only* methods proposed to learn the knowledge of a program's access pattern in memory system optimization [8, 14]. For instance, page coloring places virtual pages that may be accessed concurrently in physical frames associated with different cache blocks to reduce conflict misses. However, page coloring and its variations have a common problem: their control on data placement via virtual-to-physical address translation is coarse-grained *page* granularity. However, in 3D memory, we need fine-grained control at *cacheline* granularity in order to distribute concurrent requests across as many channels as possible to fully exploit its CLP. Simply reducing page size to cache-line size may incur prohibitively high paging overheads.

To summarize, the key insights are: 1) the hardware-only methods can achieve *fine-grained* control of data placement in physical memory but cannot support multiple access patterns; 2) software-only methods can support multiple access patterns but can only provide control mechanism at *coarse-grained* granularity due to the limitation of paging. To better exploit CLP, we need a cooperative hardware-software mechanism that can combine the best of both world which motivates our software-defined address mapping.

In this work, we propose Software-Defined Address Mapping (SDAM) — a collaborative software/hardware technique — to address these challenges. To the best of our knowledge, SDAM is the first to provide new system mechanisms that enable user program to obtain a direct control of the low-level memory hardware in a more intelligent and fine-grained manner. We apply SDAM to the case of 3D-stacking memory to better exploit its CLP by allowing the program to specify desired mappings for different data structures according to their access patterns. The proposed mechanisms include a runtime memory allocator that passes the address mapping information to the kernel and additional architectural support in the memory controller hardware for selecting the desired mapping. More specifically, the memory allocator can satisfy memory allocation call from pools of preconfigured physical memory chunks, or reconfigure free memory into the desired mapping. On a request to memory, the memory controller uses the desire mapping to calculate the hardware address of the data including which channel to use for access. The only added new hardware to the memory controller are the address mapping unit (AMU) that maps physical address to the channels/banks/rows of internal memory structure and the chunk mapping table (CMT) that is a small SRAM (67 KB) to store the AMU configuration. To guarantee the data in one chunk is associated with only one address mapping, it only requires to add two simple constraints to the allocation of virtual pages and physical frames, complying with allocation rules in Linux kernel.

To absolve programmers of responsibility for finding access pattern information to select an address mapping, we develop machine learning methods that can automatically learn the access patterns of the variables that contribute most to the external memory access and cluster these access patterns into a reduced set. The reduced number of data access patterns makes it easier for the OS to control and manage address mappings. For that, we propose chunk-based address mapping. We allocate coarse-grained regions and store a mapping for the region, then allocate space for individual data structures within the region. To expand or shrink, we allocate/free memory to/from the region in the unit of chunks. The granularity of chunk can vary, independent of page size and is typically larger than a page. The chunk size has to be selected carefully to meet a set of constraints to ensure correctness with low storage and performance overhead.

We demonstrate the benefits of proposed research methodology on real hardware comprising an FPGA-based RISC-V processor and accelerators integrated with HBM. It also includes a bootable Linux for a realistic software flow, allowing us to change Linux Kernel functions to evaluate the proposed cross-stack address mapping management. In detail, we implement a 4-core RISC-V CPU with near memory accelerators on VCU37P FPGA platforms equipped with 8GB HBM2 modules. On top of the standard RISC-V architecture, we add the proposed CMT and AMU to implement the SDAM. The software modification is based on Linux kernel 4.15 and glibc 2.26. The evaluation covers a wide range of workloads, including standard benchmarks i.e. SPEC2006, PARSEC and emerging data-intensive benchmarks on both CPU and accelerators.

The primary contributions of the paper are:

- We propose the software defined address mapping (SDAM) that employs a coarse-grained chunk-based address mapping management while achieving fine-grained data placement in hardware to fully exploit the CLP in 3D memory. The proposed method leverages knowledge of variable-level data access information in a program and associates meta-data containing address mapping information with contiguous physical memory addresses that have the same data access pattern and thus address mapping.
- We present the necessary architectural support and system software modification to enable SDAM for both CPU-only systems and systems comprising both CPU and near memory accelerators. The modifications to existing processor and system software are minimal.
- We develop machine learning methods for advanced system optimization. They can automatically identify access patterns and cluster variables that have similar access pattern to further reduce the overhead of SDAM.
- We demonstrate the effectiveness of SDAM on an FPGA-based full-system prototyping platform with full OS stack.The evaluation results show that the system implemented with SDAM achieves 1.41× speedup on standard CPU benchmarks including SPEC2006, PARSEC and 1.84× speedup on emerging data-intensive benchmarks compared to a baseline system that uses a fixed address mapping. Moreover, it achieves 2.58× speedup for near memory accelerator on the evaluated data-intensive benchmarks, compared to the case without SDAM, indicating future systems with accelerators is likely to benefit more from the proposed techniques.

2 BACKGROUND

2.1 CLP in 3D-Stacking Memory

3D-stacking memories provide significantly higher per socket peak memory bandwidth (960 GB/s) than the DDR-SDRAM family (102.4

GB/s) by employing new parallel memory architectures that vertically stack multiple DRAM dies using through-silicon-via (TSV) technology. The two realizations of 3D memory are JEDEC High Bandwidth Memory (HBM) [23] and Micron's Hybrid Memory Cube (HMC) [37]. A HBM stack comprises multiple DRAM layers organized hierarchically, where each layer is further divided into two channels, and each memory channel consists of multiple memory banks. Since TSV technology provides high pin density (1024 per HBM stack), 3D-stacking memories have significantly more independent memory channels and can serve more than 10× concurrent memory requests in parallel that traditional DDR-SDRAM-based memory.

In general, memory architecture (DDR and 3D memory) offers three levels of parallelism: channel-level parallelism (CLP), bank-level parallelism (BLP) and row-level parallelism (RLP) to exploit parallelism across channels, banks and within a row. Among them, CLP (16–32) is the highest degree of parallelism for 3D-Memory and thus is most performance-critical as memory accesses to independent channels can be served *fully* in parallel. In comparison to CLP, BLP (8) and RLP (4) have limited impact on performance since memory accesses to different banks in a channel or to different columns in a row within a bank has to be serialized due to contention for shared resources in the same memory channel.

In DDR-based memory, most prior efforts focus on utilization of RLP and BLP as the memory subsystem usually has large rows (high RLP) but few memory channels (low CLP). For instance, a modern CPU can have up to 4 DDR4 channels per socket and a row buffer size of 2 KB per channel [2]. However, one socket can integrate up to 4 HBM stacks that have a row size of 256B and each socket provides 32 independent channels [23]. Therefore, 3D memory offers 8× more CLP than DDR memory but with 8× smaller row.

Due to the key difference in architectural organization between 3D memory and traditional DDR-DRAM, it is important to maximally utilize the CLP in 3D memory when serving multiple memory requests. To better illustrate the importance of utilizing CLP for performance optimization, Fig. 1 shows that the throughput of HBM linearly with increasing the number of utilized channels to exploit CLP and sub-linearly increasing with the number of utilized columns in a row to exploit RLP.

2.2 DRAM Address Mapping

Data placement in physical memory hierarchy across channels/banks /rows is achieved via the two steps: (1) virtual to physical *address translation* (VA-to-PA translation) and (2) physical to hardware *address mapping* (PA-to-HA mapping). The VA-to-PA translation maps a large per-process virtual address space to a limited-size physical address space determined by the available hardware resources and shared by all processes. Such translation is managed by the OS's virtual memory system.

The PA-to-HA mapping[1] further transforms the flat 1D of physical addresses into an internal 3D hierarchical structure of channels, banks, and rows. The PA-to-HA mapping is typically managed by hardware (memory controller). As an illustrative example, Fig. 2

[1]PA-to-HA mapping is also refereed to as address mapping in the paper

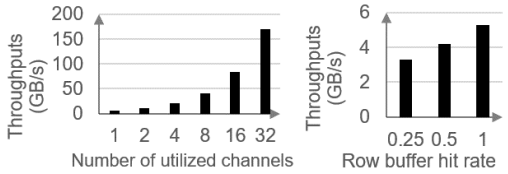

Figure 1: Comparing HBM throughput for different number of channels and row buffer hit rate

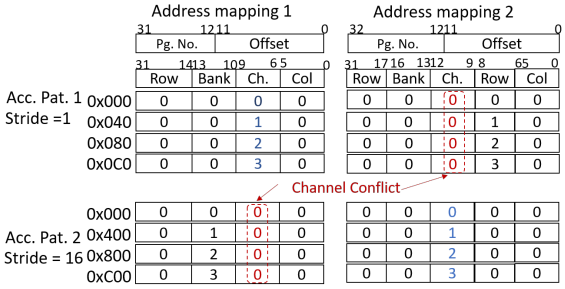

Figure 2: Illustration of channel conflicts (in red) for different access patterns and address mappings. The access granularity is 64B (cache-line size of RISC-V architecture).

compares two different address mappings: (1) a 32-bit physical address is split into a 18-bit row address, a 4-bit bank address, a 4-bit channel address, and a 6-bit column address; (2) the 18-bit row address is splitted into two parts and the 3 LSBs are put between the channel address and the column address. The address mapping determines the CLP utilization and is highly dependent on the access patterns. In the example, when accessing data with a stride of 1 (streaming), mapping 1 allows consecutive memory access to be evenly served by different channels and to exploit the CLP better than mapping 2. With the same address mapping, accessing data with a stride of 16 leads to severe channel contention, as only 1 out of 16 memory channels is used. Similarly, the mapping 2 can better exploit the CLP for data access with a stride of 16, but causes channel conflict for streaming access.

2.3 Address Mapping Mechanisms

Hardware-only methods: The first class of methods to control data placement across channels/banks/rows to improve bandwidth utilization are hardware-based via the direct control of PA-to-HA address mapping.

Most commercial computer systems [20, 22] use a boot-time configured PA-to-HA mapping. During the device discovery phase, the BIOS determines the address mapping based on the DIMM configuration. Memory accesses to consecutive cache blocks are uniformly distributed to different memory channels (channel interleaving), resulting in higher throughput for streaming data access. However, such *fixed* and *global* address mapping may lead to imbalanced channel utilization and performance degradation for non-streaming access patterns.

To illustrate these problems, we construct a synthetic workload that reads data with different strides from HBM using Xilinx VU37P

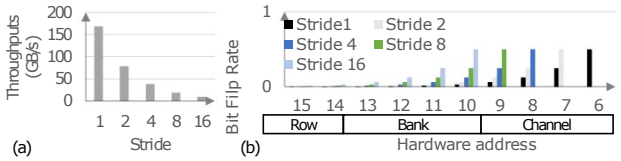

Figure 3: (a) HBM throughput with different stride using the default address mapping; (b) Bit flip distribution for different stride using the default address mapping

FPGA (following the same experimental setup as that in Section 7). As in Fig. 3, using the default HBM address mapping defined in the Xilinx HBM controller IP [23], the throughput drops sharply by 20× when increasing the stride from 1 to 16 in the unit of cache-line size. That is because only a subset of the memory channels are utilized. In the worst case, e.g., a stride of 32, only one channel is utilized. To improve channel utilization (and CLP), we need to make the PA-to-HA mapping adapt to the different data access patterns while keeping fine-grained control to distribute consecutive memory access to distinct channels.

Software-only methods: The software-only approaches use OS support [8, 9, 14, 29, 42] to more intelligently place data in physical memory by changing the VA-to-PA translation in virtual memory (VM) according to the access patterns of data structures. These works attempt to find variables/data structures in a program that may be accessed concurrently and use the information to direct the operating system's VM page mapping strategy to place them to physical pages that do not contend for the same cache line, thereby reducing conflict misses. By effectively leveraging high-level program semantic information, these methods can better predict the program's behavior in data access and achieve higher and more deterministic cache performance. However, these methods assume fixed hardware and allow controlling allocation to place data in hardware (caches) at coarse-grained *page* granularity [8, 9, 14, 42]. However, as shown in Fig. 2, to exploit CLP for two different access patterns (stride-1 and stride-16), the page-based data placement becomes insufficient. We need more fine-grained control at *cache-line* granularity to distribute concurrent memory requests across as many channels as possible to fully exploit its CLP.

To summarize, the key limitations are: 1) the hardware-only methods can achieve *fine-grained* control of data placement in physical memory but cannot support multiple access patterns; 2) software-only methods can support multiple access patterns of a program but only provide control mechanism at *coarse-grained* granularity due to the limitation of paging. To better exploit CLP, we need a cooperative hardware-software mechanism that can combine the best of both worlds, which motivates our software-defined address mapping.

3 MOTIVATING EXPERIMENTS

In this section, we present three experimental studies to motivate our proposed research approach. We use the same experimental setup as that in Section 7 using a Xilinx VU37P FPGA [46] with two HBM2 stacks which have 32 channels in total.

Experiment 1: In Fig. 3(a), we demonstrate the relationship between data access patterns and memory throughput. We construct

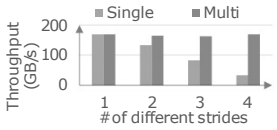

Figure 4: Throughput comparison of using single and multiple address mapping for workloads with mixed access patterns

Table 1: Summary of variable-level statistics

	Benchmark	# of Var.	# of Major Var.	Avg. Major Var. Size (MB)	Min. Major Var. Size (MB)
SPEC2006	perlbench	7268	1	910	910
	bzip2	10	10	32	4
	gcc	49690	34	59	4
	mcf	3	3	1215	953
	gobmk	43	5	8	7
	hmmer	84	10	6	4
	sjeng	4	4	60	54
	libquantum	10	7	212	4
	h264ref	193	8	24	7
	omnetpp	9400	65	3	1
	astar	178	38	1.8	9
	xalancbmk	4802	4	230	78
PARSEC	bodytrack	220	12	212	36
	cenneal	17	9	365	69
	dedup	29	15	215	12
	ferret	109	22	65	23
	freqmine	60	9	215	37
	streamcluster	35	9	234	68
	vips	892	25	125	36

three synthetic workloads, which read data from memory with different strides. We apply the same method as prior hardware-only approach [1] to use bit flip rate as a metric to select desired address mapping. We pick bits of physical addresses that vary the most to be mapped onto the channels to distribute accesses across the most number of channels. Fig. 3(b) shows the normalized bit flip rate in the stream of memory access. With increasing stride, the flip rate peak moves to the left, indicating the optimal address mapping for different access patterns varies.

Observation 1: The address mapping needs to adapt to different memory access patterns to best exploit the CLP.

Experiment 2: The second experiment shows the impact of mixing multiple data access patterns. Fig. 4 shows the throughput of the four strides from experiment 1 with the globally best mapping (maximum bit flip rate) and with a separate mapping for each stride. In case-1, we apply the same method as experiment 1 to select the optimal address mapping based on overall bit flip rate when running the workload mix comprising different concurrent memory accesses. Alternatively, in case-2, we independently choose the optimal address mapping for each access pattern and measure the aggregated throughput. The results show that a global address mapping cannot deliver the best performance as compared with the case of independently selecting address mapping for each individual access pattern.

Observation 2: It is important to capture the detailed program's access behaviors i.e., the diversity of the access patterns to guide address mapping selection for higher CLP.

Experiment 3: In the third experiment, we profile SPEC2006 and PARSEC workloads to identify the number of unique data structures that may have different access patterns. The detailed profiling process is described in Section 5. As illustrated in Table 1, we run 19 applications to collect variable-level statistics of SPEC2006 and PARSEC. As defined in [24], a variable is the reference symbol in

the program for a piece of allocated memory and is the granularity of memory management from programmer's view. These applications show highly diverse distributions in the number of variables (ranging from 3 to 49k). While the programs have a wide range and often large number of variables, when we focus on those that comprise 80% of references – *major variables* – we find that they are fewer in number and large in size. This makes it feasible to track these variables in hardware and provide distinct address mappings for each one if necessary.

Observation 3: A limited number of *major variables* contribute to most of the external memory accesses and have large memory footprints.

4 SOFTWARE-DEFINED ADDRESS MAPPING

From Section 3, we learned that a limited number of major variables in a program have a large memory footprint and contribute to the vast majority of external memory accesses in hardware. It inspires our SDAM: Applications provide access pattern information to the OS, which configures large chunks of memory for the aggregate allocations reached by a variable. For efficiency, the OS maintains pools of memory for each address mapping, and only reconfigures when memory is reclaimed or more memory with a specific mapping is requested. The hardware provides a table to record the address mapping for each chunk of memory.

Coarse-grained Chunk-based Address Mapping. Observation 3 in Section 3 inspires our chunk-based address mapping, which manages address mapping at a coarse-grained chunk granularity (2MB in our case). More specifically, we allocate coarse-grained regions and store mapping for the region, then allocate space for individual data structures within the region from one or more processes. And, to expand or shrink, we allocate/free memory to/from the region in the unit of chunks. The chunk size can be selected, independent of page size, to meet a set of constraints to ensure correctness with low storage and performance overhead, described later in this section. In chunk-based address mapping, we map a pool of data with the same (or similar) access patterns to chunks during the allocation of virtual pages and physical frames. Each chunk consists of contiguous physical pages containing data with the same access pattern. The kernel maintains a chunk mapping table (CMT) that store address mapping information for each chunk. At runtime, the memory controller retrieves the mapping on each reference from CMT to determine how to use bits from the address to select channel, row, and bank indexes. Therefore, at high level, chunk-based address mapping is a collaborative software/hardware technique that improves PA-to-HA address mapping but does not need any modification to existing VA-to-PA translation.

The required modification to system software is straightforward. To coalesce data with the same access pattern into chunks, we add additional constraints to the allocation of physical frames and virtual pages in software. For physical frame allocation, SDAM configures all physical frames in one chunk have the same address mapping. For virtual allocations, it uses pages from the chunks with the same address mapping. For this, we modify the malloc() function to take the desired address mapping as an additional argument, and store the address mapping as meta-data for allocated virtual pages and chunks. Malloc will allocate pages with the desired mapping,

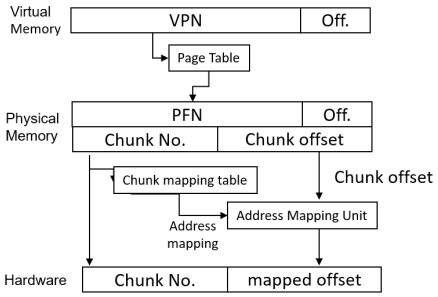

Figure 5: Chunk-based address mapping management

and can call the kernel for more memory with the desired mapping. The kernel will allocate pages from any available chunks with that mapping, and if none exist, create a new chunk for the desired mapping. The detailed modification to the system software can be found in Section VI-A. In addition, in Section VI-B, we discuss a mechanism to statically and dynamically analyze data structure reference pattern, and apply machine learning to determine a set of variables with distinct reference patterns to further reduce the overhead of SDAM.

The chunk-based address mapping has two steps: (1) looking up the address mapping based on the PA of chunks, (2) calculating the HA based on the PA and address mapping and using HA to place the cache blocks across channels/banks/rows in 3D memory. As in Fig. 5, the chunk mapping table stores the address mapping for each chunk. Compared to the page table in virtual memory system, the CMT is much smaller, since (1) the physical memory space is much smaller than the virtual memory space and is globally shared by all the processes and (2) the chunk size is much larger than typical page size. (3) 3D-stacked memory is often much smaller due to physical constraints. The physical address of chunks is divided into two parts: chunk number and chunk offset. The chunk number is used as an index to look up the corresponding address mapping in the CMT and the chunk offset is applied as inputs to the hardware address mapping unit, which calculates the HA based on the address mapping. Finally, according to the HA bit field information, memory controller performs data placement in internal structure of physical 3D memory.

The proposed chunk-based address mapping has low implementation overhead. First of all, the only new hardware is the address mapping unit (AMU) that converts the PA to HA based on the selected address mapping. Secondly, we can use a small table to store the meta-data associated with each access pattern (address mapping), resulting in low storage overhead and fast CMT look-up. Finally, the modifications to the virtual and physical memory allocators are minor.

Next, we discuss a few properties of SDAM.

Functional correctness guarantee: As explained previously, the proposed method keeps standard virtual memory unchanged to ensure correct VA-to-PA translation. As such, we only need to guarantee the correct PA-to-HA mapping, i.e., one PA can map to only one HA or verse versa. To ensure correct PA-to-HA mapping, we need to consider two cases: inter-chunk and intra-chunk: (i) In case of inter-chunk mapping, we keep the chunk number (high order bits of the physical address) unchanged during PA-to-HA

address mapping. Since the chunk number is directly copied from the MSBs of the physical frame number (PFN), it guarantees no inter-chunk mapping conflict. (ii) For intra-chunk mapping, all addresses in a chunk use the same and invertible address mapping function, which can guarantee 1-to-1 mapping from PA to HA. A rigorous mathematical proof for that can be found in [1].

Chunk size: To determine the chunk size, we first consider a hard constraint. Since address mapping only applies to chunk offset, so chunk size should large enough to cover variety strides. Also, since we need to keep track of chunks using a mapping table, to keep a low storage overhead, we would prefer a large chunk size. However, similar to pages in VM, larger chunk sizes may cause internal fragmentation, as the free space in one chunk cannot be allocated to others due to the address mapping constraint. In our implementation, we choose a chunk size of 2MB to balance storage overhead and internal fragmentation based on the following overhead analysis.

(i). Storage overhead: The 2MB chunk size leads to low storage overhead. The physical memory space is much smaller than the virtual memory space and is globally shared by all the processes, so the total number of chunks in physical memory is limited. In our system with 8GB HBM, we only have 4096 chunks, that is much less than the number of physical frames. Moreover, as the number of data access patterns is significantly reduced after applying machine learning-based optimization (discussed in Section 6.2), we only need to store 1 byte to encode them. As a result, our CMT size consumes only 68 KB. Due to the compactness of CMT, it can be implemented using a small fast on-chip SRAM with negligible latency compared to the HBM access latency. More details can be found in Section 5.

(ii). Fragmentation: The 2MB chunk size does not suffer from the same internal fragmentation issue as that in huge page. Internal fragmentation at the chunk level in SDAM is bounded by the number of access patterns rather than the the number of chunks as that in huge page. Our system supports up to 256 access patterns, which is confirmed to be sufficient in our evaluation. In the worst memory allocation pattern, we would only waste 256 chunks (6.25% of the total number of chunks, 256/4096=6.25%) due to the internal fragmentation. Furthermore, since SDAM does not need to expand the number of chunk groups for larger memory capacity, the overhead of non-allocatable memory caused by internal fragmentation would be even smaller for larger HBM capacity. Finally, SDAM uses paging for memory allocation that does not have external fragmentation.

The chunk-based address mapping does not introduce security vulnerabilities. Nevertheless, it can be exploited to mitigate row hammer attacks. As each chunk consists of a large number of contiguous rows within a bank, we can mitigate the row hammer attack by adding guard rows to the sensitive data to ensure the strong physical isolation between data belonging to different security domains, following the same methodology in [7]. More detailed study on extending SDAM to address security challenges will be our future work.

5 ARCHITECTURAL SUPPORT

5.1 Overview

To implement SDAM, the required hardware modifications to existing processor are relatively minor. There is no need to modify the architecture/micro-architecture of CPU cores, e.g., ISA, TLB, page walker. The only modification is to add two dedicated hardware components in the memory controller: an address mapping unit (AMU) and on-chip SRAM to store the CMT. (i) The AMU contains a simple crossbar to support arbitrary address remapping. (ii) To achieve compact storage, we use two-level CMT that stores the indices of the address mapping to associate with chunks and address mappings separately. As shown in Fig. 6, for each external memory access, the physical address is divided into two parts: the chunk number (MSBs) and the chunk offset (LSBs). The chunk number is used to index the CMT to find the corresponding entry for address mapping. Then, the AMU maps the chunk offset from the physical address to the hardware address. Finally, the chunk number and the mapped chunk offset are sent to the memory controller. Next, we describe the detailed design of the AMU and the CMT.

5.2 Address Mapping Unit

The address mapping unit (AMU) maps the chunk offset in the physical memory space to that in the hardware address space.

We choose to implement the bit-shuffle mechanism in the AMU, to rearrange the address bits in an arbitrary order. From the discussion in [1] and our evaluation in Section 7, we confirmed that the bit-shuffle is flexible enough to support different address mappings for the workloads of interest and has low implementation overheads. As shown in Fig. 6, AMU implements the bit-shuffle using a simple crossbar, which is an array of switches. In the crossbar, the bit-shuffle allows only one closed switch in each column. The crossbar can be configured by the address mapping information stored in the CMT. Since each input to the crossbar is a single bit, the crossbar requires only n^2 switches, where n is the size of the chunk offset. Using a 2MB chunk size and a 64B cache-line size, our implementation has a 15-bit chunk offset. Thus, the AMU only adds 2 % area to our RISC-V CPU. It is negligible compared with the performance benefit from improved CLP utilization. More evaluation are presented in Section 7.

5.3 Chunk Mapping Table

The CMT stores the meta-data in a compact table containing address mapping information for each entry. To keep storage overhead low, the CMT stores the address mapping in two tables. As in Fig. 6, entries in the first table stores the index of the address mapping and those in the second table store the actual address mapping that is used to control the AMU. Although the AMU supports a large number of address mappings, only a few of them are used concurrently. Using these two tables can reduce the storage requirement since there are more chunks than the concurrent address mapping in the program.

As discussed in Section 5.2, the crossbar in the AMU allows only one closed switch in each column. Hence, the crossbar configuration for rearranging n chunk offset bits, requires only n integers, that represent the locations of closed switch in columns. Since there are n switches in each column, each integer should have $log_2(n)$ bits. In our implementation, as the chunk offset is 15-bit, the configuration of the AMU requires $15 \times log_2(15) \approx 60$ bits. Considering the 2MB chunk size used in this work and a system configuration that has 128GB 3D-stacking memory per socket, the chunk-mapping

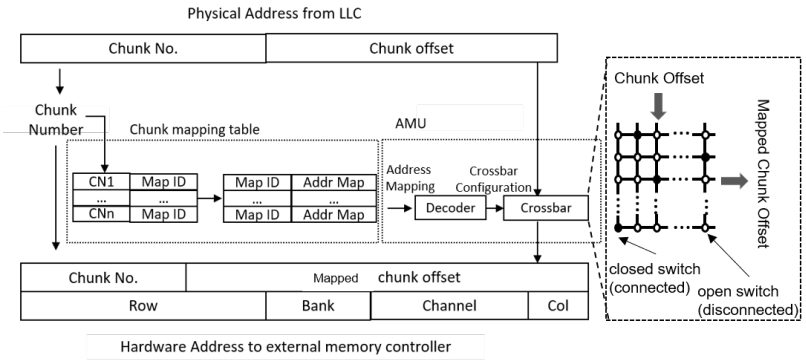

Figure 6: Hardware modification

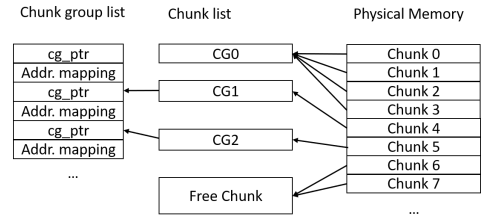

Figure 7: Chunk allocation

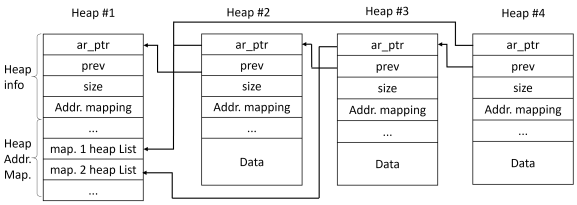

Figure 8: Multi-heap memory allocation

table has 64k entries. To support 256 concurrent address mappings, which is confirmed to be sufficient in our evaluation (Section 7), each entry in the first table has only $log_2(256) = 8$ bits. The two-level table design only requires 67.94 KB (64k entries×8 bits/entry+ 256 entries × 60 bits/entry) in storage. In comparison, storing the meta-data using a flat table would require 491 kB (64k entries × 60 bits/entry). Due to the compact storage of the two-level address mapping table, we can implement CMT using a small fast on-chip SRAM with 6 ns latency that is negligible in comparison to the HBM access latency (> 130ns).

6 SOFTWARE SUPPORT

6.1 Address-Mapping-Aware Memory Allocation

The chunk-based address mapping requires modest modifications to the system software. For both virtual memory and physical frame allocations, rather than using a single allocator, we separate memory based on the address mappings. Then, we can have an allocator for each mapping, and provide a way to move memory between mappings.

Virtual Memory Allocator. We extend the malloc() in glibc to associate heaps with address mappings and leverage the standard heap memory allocation procedure for each memory allocation. As each heap is associated with one address mapping, we only need to use the address mapping to select the heap, and use the existing heap code to handle the allocation/deallocation within heap. As in the standard malloc() in glibc, the proposed mechanism keeps separate memory pools (arena in glibc) for each thread to reduce the lock contention [16].

As in Fig. 8, in the first heap, we maintain an array to track the heaps assigned by address mapping, i.e. heap-mapping array, and also an array to track the address mapping used by the process. We add a new API add_addr_map(), that adds a new address mapping and returns the ID for this address mapping. Also, we add the ID of address mapping as an optional input argument of the malloc() function. Each malloc() call starts with checking if any heap matches its address mapping. If it fails, a new heap is created and attached to the heap list in the first heap. Otherwise, it will check if there is enough space. If not, it will also create a new heap. Then it will allocate the memory in the heap with the matched address mapping. Since addresses allocated to heaps are page-aligned and heaps allocate/free memory independently, this method can guarantee that each page contains data that only have the same address mapping. To free the allocated heap space, the free() function compares the base address of the variable with the ar_ptr and size (Fig. 8) to find the corresponding heap.

Physical Page Allocator. As shown in Fig. 7, in the physical memory space, we manage chunks as chunk groups. Each chunk group contains chunks with the same mapping (and thus the same access pattern). At the beginning, all chunks are in the global free-list that contains all the unused chunks. Then, we allocate/free chunks to/from the chunk group associated with different address mappings as needed.

Similar to the modifications to the malloc(), we add the address mapping ID as an argument of the mmap() that allocates the physical page. For each mmap() call, if there is not enough memory in the corresponding chunk group, it acquires one or more chunks from the global chunk free-list and writes the chunk index and address mapping to the hardware CMT. The proposed method is compatible with on-demand paging. To support the on-demand paging, for each modified mmap() call, we store the address mapping ID in the *vm_area_struct* in the kernel and move the proposed chunk-based physical page allocator to the page fault handler. The page fault handler can allocate the physical page based on the address mapping ID.

We rely on the original Linux buddy allocator to free the chunks. If all the blocks within one chunk are freed by the user program, the chunk buddy allocator sets all the bits in the block to 0 and adds it to the chunk free-list.

6.2 Address Mapping Selection

For programs with simple repetitive data access such as element size and stride, programmers can identify the access pattern and select the address mapping directly from the source code.

In more complex scenarios, to absolve programmers of responsibility for finding access pattern information to select an address mapping, we develop a method that automatically i) finds the variables that contribute most to the external memory access (or referred to as *major variables*), ii) applies machine learning to identify the access patterns of these major variables, and iii) further clusters the access patterns into a reduced set to reduce the storage overhead of the meta-data. Depending on the number of identified major variables of co-run applications and the hardware constraint such as the size of the CMT, we provide additional flexibility of making quality-time trade offs for selecting address mapping, i.e. fast runtime using K-Means in which the clustering quality is likely constrained vs. slow runtime using (DL)-assist K-Means for high quality. After profiling and training, the CMT chunk table will be updated over time.

First, we use profiling to find major variables and associate each major variable with its corresponding physical memory address (PA) trace. At compile time, we use gcc to create a table that maps program counters (PC) to the variables referenced at that PC. Then, we run the application and collect the physical address of each external memory access and its corresponding PC value using the profiling tools as described in [48] (identified by call-stack matching [24]). The call-stack matching has two passes. In the first pass, we run the program with a modified malloc() by preloading a shared library that intercepts all heap memory allocation routine and collects allocation call stack. Once we have the call-stack information, we rerun the program with the profiler and find the allocation sites corresponding to a memory reference by matching the call stack. In this way, we can separate the collected PA trace into sub-traces associated with each allocation site(variables) and the address mapping are learned for each variable.

Based on the PC, we can associate each variable in the source code with a corresponding PA trace.

For programs known to have few major variables or in situations with constrained hardware that can only support a small number of mappings, we can use classical machine learning K-Means to identify the few best (most representative) reference patterns with which to create address mappings. In particular, we calculate the bit-flip rate vector BFRV below:

$$\text{BFRV} = [\text{bfr}_1, \text{bfr}_2, \ldots, \text{bfr}_m],\ \text{bfr}_i = \frac{\sum_{j=1}^{n-1} \text{bit}_{i,j} \oplus \text{bit}_{i,j+1}}{n}, \quad (1)$$

where m is the total number of address bits and bfr_i is the i-th element of the BFRV, n is the number of memory accesses in the PA address trace and $\text{bit}_{i,j}$ is the i-th address bit of j-th memory access in the trace associated with each major variable. We then apply K-Means clustering to minimize the clustering loss [31]:

$$\mathcal{L}_{\text{cluster}} = \sum_{i=1}^{k} \sum_{\text{BFRV} \in S_i} ||\text{BFRV} - \mu_i||^2, \quad (2)$$

where k is the number of clusters, S_i is a cluster containing BFRVs with similar access pattern and μ_i is the averaged BFRV in each cluster which determines the address mapping of these variables. The bits with higher bit-flip rate are used for channel address while the bits with lower bit-flip rate are mapped onto banks and rows.

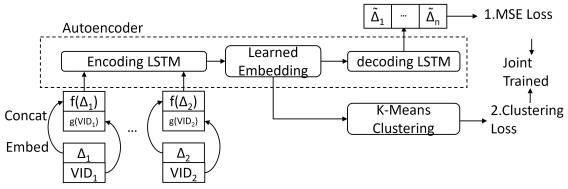

Figure 9: The embedding LSTM model.

For complex programs with more major variables, K-Means may be less effective, as the bit-flip rate vector becomes a poor representation for clustering when further increasing data set size. To improve the clustering quality, we propose to learn a clustering-friendly representation (embedding) for each application from the trace using Autoencoder [4] and classifies the bit-flip patterns using the learned embedding. To the best of our knowledge, this work is the first to use deep learning to improve the quality of address mapping selection, which is an *unsupervised* learning problem since there is no groundtruth (label) of address mapping. In comparison, previous work [3, 18] focus on applying deep learning for prefetching which is a *supervised* learning problem since the groundtruth of a prefetching decision is the address of the following memory access. We summarize the key steps of the DL-assisted address mapping selection using the embedding Long Short Term Memory (LSTM) model below.

(1) As shown in Fig. 9, the input is a sequence of (Δ, VID) pairs, where Δ is the address difference (XOR result) between two consecutive memory accesses and VID is the index of the variable identified by gcc. Δ and VID are separately embedded (mapping of a discrete–categorical–variable to a vector of continuous numbers) and the embedding is concatenated and fed as an input to an LSTM-based auto-encoder [18] of which the hyper-parameters are summarized in Table 2. We first train the auto-encoder by minimizing the reconstruction loss function, defined as follows:

$$\mathcal{L}_{reconstruct} = \mathcal{L}(\Delta, \widetilde{\Delta}, W) = \sum_{i}^{n} \sum_{j}^{m} |\widetilde{\Delta}_{i,j} - f_w(\Delta_{i,j})|, \quad (3)$$

where m is the total number of address bits, n is the number of memory accesses and $\Delta_{i,j}$ is the j-th address bit of i-th memory access in a Δ trace.

(2) We then apply K-Means on the *learned* embedding vectors and continue the training by minimizing the MSE loss and clustering loss jointly using the loss function $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruct}} + \lambda \mathcal{L}_{\text{cluster}}$. This DL-assisted K-Means on the high-dimensional embedding vector (256-dim) compared to the K-Means on the bit-flip rate vector (15-dim), improves the clustering quality as the learned embedding can meaningfully represent categories in the transformed space [33].

(3) Once we have a cluster, we use the bit flip rate to select the desired address mapping for all variables in the cluster. The highly flipping bits correspond to frequent accesses in a short time and are mapped onto channel address bits to best exploit the CLP, while the less frequently flipping bits are mapped onto banks and rows.

(4) We add the ID of the selected address mapping to the input argument of the malloc() function as in Section 6.1.

Table 2: Training hyper-parameters

Network size	256x2 LSTM	**Learning Rate**	0.001
Steps	500k	**Sequential Length**	32
Embedding Size	256	λ	0.01

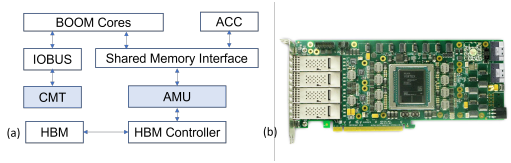

Figure 10: (a) Diagram of the prototyping system; (b) Photo of the FPGA prototyping platform with integrated HBM devices

We note that, profiling is done *offline*, *once* for each application. This will not degrade the runtime performance when input changes. As we will show in our evaluation (Section 7.4), similar speedup can be achieved even when different inputs are used for profiling and execution. For software that runs many times or has a long execution time, the profiling cost added is incidental. Moreover, in the context of software development, the added cost can be further amortized since the profiling result can be reused across variations of the program as long as the data structure and memory allocation site do not change. In practice, profiling-guided optimization (PGO) has been widely deployed by different types of commercial software such as browsers, OS kernel, and datacenter applications. Moreover, a similar high level motivation for using PGO to improve system performance has been used in prior work [3, 18] that applies profiling to improve the performance of software prefetching. As discussed earlier, SDAM differs from these work in several key aspects. In addition to the fundamental machine learning method (supervised vs unsupervised), SDAM is developed to improve data placement in 3D memory instead of cache performance. In Section 7.4, we will show more detailed results to compare the profiling time and quality. In practice, the address mapping selection method can be judiciously determined based on the number of major variables and the number of clusters (K) and the CMT size to balance profiling complexity and quality and make the optimal quality-time trade off.

7 EVALUATION

7.1 Prototyping Platform

To accurately capture the detailed HBM-specific behaviors and comprehensive interactions between OS and the hardware, we evaluate our design on an open source full system FPGA prototyping platform MEG [48] with integrated HBM devices and full OS stack. Our baseline system comprises a BOOM RISC-V processor [10] and shared-memory interfaces for near-memory accelerators on the AlphaData 9H7 FPGA board equipped with an Xilinx VU37P FPGA and an in-package 8GB HBM2 memory [45]. It also supports Linux, allowing us to evaluate our modifications to the functions in glibc and Linux.

Specifically, the BOOM CPU consists of four 64-bit out-of-order cores with 64 KB L1 caches that run at 200 MHz. Multiple application-specific accelerators generated by SystemVerilog are integrated with the CPU through shared memory interface provided by MEG platform. The Xilinx VU37P FPGA integrates two HBM GEN2 stacks

Table 3: FPGA resource utilization

	LOGIC	SRAM
Boom Core	91.8%	88.0%
HBM Controller	7.5%	10.2%
AMU	0.5%	0%
CMT	0.2%	1.8%

Table 4: LOC changed

Feature	LOC changed
VM allocator	131
PM allocator	97
Driver	98
Miscellaneous	33

with 32 independent memory channels. Due to the hardware limitation of our prototyping platform, in this work, we did not evaluate more comprehensive memory subsystem which comprises heterogeneous memories (e.g. DDR, HBM, NVM). A large body of prior work have explored the topic [12, 26, 32, 35, 38, 42] and provide mechanisms to migrate performance critical data from slow DDR memory to fast HBM. These works are orthogonal to ours and can benefit from our technique when combing them.

To implement our design, the modified hardware is highlighted in Fig. 10. We add the AMU and the CMT (implemented in SystemVerilog) between the HBM2 controller and the Boom core. The AMU is attached to the memory bus, and its upstream is the last level cache (LLC). The AMU maps the PA to HA and sends the HA to the memory controller. We duplicate the AMU eight times to guarantee the peak HBM bandwidth can be achieved (4 billion HBM access (64B) per second). In the real CPU implementation, the duplication of AMUs is unnecessary as the CMOS-based logic is much faster than the FPGA-based logic (4GHz vs 500 MHz) The CMT is attached to the I/O bus of the CPU, so that OS can modify the content of the table through memory-mapped IO. The design is implemented using Xilinx Vivado 2018.4 and Xilinx AXI-HBM IP 1.0. The resource utilization including the duplicated AMU is summarized in Table 3. The two newly added hardware components (AMU and CMT) have negligible area overhead compared to the baseline system.

We implement the software modification in linux 4.15 and glibc 2.26. The lines of code changed split up by functionality are shown in Table 4.

7.2 Workloads

In the experiments, to perform comprehensive evaluation, we select a broad set of benchmarks, including synthetic memory-access benchmark, standard benchmarks i.e. SPEC2006 and PARSEC, as well as representative emerging data-intensive benchmarks in three important domains covering large-scale graph processing, in memory data analytics, machine learning and information retrieval. In addition to the evaluation on CPU, we also evaluate the performance gain of offloading data-intensive workloads to hardware accelerators which becomes increasingly important in data center and HPC. Below are the details of evaluated benchmarks.

Synthetic benchmark: We use data copy with different strides as a synthetic benchmark. The size of each data copy is the same as the cache-line size of the RISC-V processor (64B).

SPEC2006: We studied all 12 integer applications [19].

PARSEC: PARSEC [6] is a popular benchmark for evaluating shared-memory multi-processor. The benchmark covers a wide range of memory behaviours including working set size, locality and external memory access. We studied all 7 applications.

Data-intensive benchmarks: In addition to the synthetic and standard benchmarks on CPU, we also evaluate 8 representative

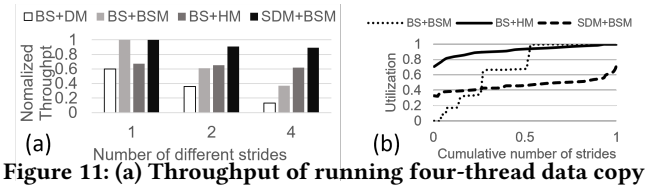

Figure 11: (a) Throughput of running four-thread data copy with different stride, normalized to the peak streaming throughput. (b) Distribution of HBM bandwidth utilization for 64 different strides using BS+BSM, BS+HM and SDM+BSM.

data-intensive benchmarks written in both C++ and SystemVerilog for both cases of running on CPU alone and offloading to accelerators, including large-scale graph processing (Breadth-First Search [47], PageRank [21], Single-Source Shortest Path [34]), in-memory data analytics (Hash Join [5], Merge-Sort Join [43]), machine learning and information retrieval (K-Means [31], HNSW [25] and IVFPQ [25]). The custom accelerators allow concurrent execution by adding more pipeline stages and parallel compute units. Thus, they can generate more concurrent memory accesses to the external memory than CPU, and its performance is more sensitive to CLP utilization compared to the CPU workloads.

7.3 System Configuration

We compare the proposed SDAM with three baseline systems that adopt the hardware-only address mapping approach proposed in prior work [1] [30] as a default address mapping (DM). We also include two more complex address mapping schemes: bit-shuffle address mapping(BSM) and hashing-based address mapping (HM) [30]. The bit-shuffle approach rearranges the order of address bits according to the distribution of the bit flip rate collected from profiling. The hashing-based approach maximizes the entropy in the channel bits by XORing a number of address bits. As summarized in Section 2.3, software-based solutions use fixed hardware and control the data allocation at page granularity. Therefore, the three baseline systems which use a single static address mapping can represent software-based solutions.

Baseline system + default address mapping (BS+DM): A single, fixed address mapping as defined in the Xilinx HBM controller IP [23]) is applied globally to all applications.

Baseline system + bit-shuffle address mapping (BS+BSM): This configuration still uses a single address mapping for all applications but trends to optimize the address mapping selection according to the profiling results. In particular, we collect physical addresses of 500 million of cache misses per benchmark and calculate the bit flip rate for the workload mix combining all benchmarks. Then, we select the optimal address mapping based on the bit flip rate where the highly flipping bits are mapped onto channels to best exploit the CLP, while the rest are mapped onto banks and rows.

Baseline system + hashing-based address mapping (BS+HM): Similar to BS+DM and BS+BSM, this configuration uses a single address mapping for all applications but applies a different optimization method, i.e., using hashing [30] for address mapping selection. The selected hash function is capable of harvesting entropy from *many* and *randomly* selected address bits and concentrating the entropy into channel bits. In comparison to BSM, HM does not rely on profiling. In this configuration, we refer to the method in a recent work [30] which provides a good balance between implementation complexity and performance gain. In our study, we found theoretically perfect hashing function leads to marginal speedup (<3%) over [30] at the cost of significantly increased overhead. We defer more comprehensive hashing methods to our future work.

Software-defined mapping + bit-shuffle address mapping without machine learning (SDM+BSM): This configuration uses the proposed SDAM (with the AMU and the CMT and running modified Linux and glibc). For each process, it uses a single address mapping chosen using bit-flip rate.

Software-defined mapping + bit-shuffle address mapping with machine learning: K-Means (SDM +BSM+ML) and DL-assisted K-Means (SDM+BSM+DL): These two configurations evaluate our complete design. As compared with the case of SDM+BSM that selects one address mapping for one application, SDM +BSM +ML and SDM+BSM+DL select the optimal address mapping for each individual variable within an application using the K-Means-based and DL-assisted K-Means address mapping selection, respectively.

For systems of BS+BSM, SDM+ BSM, SDM+ BSM+ML, SDM+BSM +DL that require profiling, we compare the results using different program input for profiling and evaluation. In detail, for SPEC2006 and PARSEC, we use the training dataset for profiling and test dataset for the evaluation. Both datasets are selected by the script provided by the benchmark suite (e.g. runspec). For Graph Workload, we generate different graphs using graph500 generator with different seed for profiling and test. The scale is 20 and edge factor is 16.

7.4 Results

In the synthetic benchmark, we run four threads for data copy and vary the strides. Since the optimal address mapping can be derived from the strides directly, we do not need to use profiling. As shown in Fig. 11(a), when there is only one access pattern (number of strides = 1), both BS+BSM and SDM+BSM achieve the maximum throughput and outperform the other two baseline systems (BS+DM and BS+HM). It demonstrates the bit-shuffle approach can be effective if the address mapping can be optimally selected according to a single access pattern. However, in more complex cases of increasing concurrent data accesses with different strides, we observe a significant performance degradation for BS+BSM, as a fixed and global address mapping becomes less effective in capturing the diversity in access behaviors. In comparison, the performance of BS+HM is nearly kept constant across all the three cases but is still worse than SDM+BSM. The reason for the constant performance is that the hash function used by BS+HM is optimally selected to be less sensitive to different access behaviors, as it takes many address bits as input to cover a majority of access patterns [30]. We plot the distribution of the CLP utilization of different strides (1 to 64) using BS+BSM, BS+HM and SDM+BSM in Fig. 11(b). For better illustration, we sort the CLP utilization in ascending order. We can see that the BS+HM tends to maximize the averaged CLP utilization for a large number of access patterns statistically, but the hashing function cannot cover all possible ones and thus for some cases may lead to underutilized CLP compared to BS+BSM. In contrast,

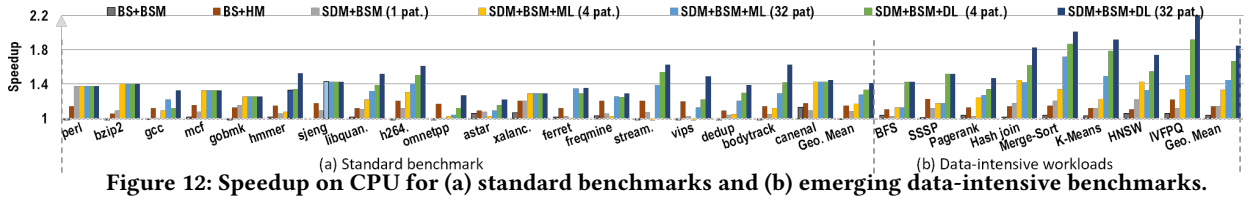

Figure 12: Speedup on CPU for (a) standard benchmarks and (b) emerging data-intensive benchmarks.

SDM+BSM try to select the optimal address mapping deterministically for each access pattern. Thus, as shown in Fig. 11(a) and Fig. 11(b), the proposed SDM+BSM consistently outperforms the other three systems for all the strides and the performance benefit of SDM+BSM grows with respect to the other three as data access patterns become more complex. These results validate the effectiveness of SDAM as compared with the hardware-only global address mapping in the baseline systems.

We further provide evaluation results for the standard CPU benchmarks. In Fig. 12, we compare the speedup of BS+BSM, BS+HM, SDM+BSM, SDM+BSM+ML and SDM+BSM+DL against the baseline BS+DM using *different* inputs for profiling and evaluation and perform cross-validation. For SDM+BSM+ML and SDM+BSM+DL, we include results using different numbers of clusters (4 and 32). Using 4 clusters per application represents the case in which several variables may need to share the same address mapping when there is a large number of co-run applications but only a limited number of chunk table entries can be used for each application. In comparison, in the case of using 32 clusters per application, each application may acquire enough chunk table entries so that each major variable can have its address mapping, resulting in improved accuracy.

As expected, the BS+BSM only has an average speedup of 1.01× over the baseline BS+DM. For some benchmarks e.g., perl and stream, the SDM+BSM shows worse performance than BS+BSM. It is because the global address mapping cannot effectively capture the different access patterns within/across applications, even when address mapping is optimally selected. It is not practical to find a single address mapping to benefit all applications. The SDM+BSM achieves more performance improvement (1.08×, 1.09×) than BS+DM and BS+BSM, as it provides application-dependent address mapping (one address mapping for one application) to better adapt to different access patterns.

In addition, we observe the same trend as that in the synthetic benchmark. The BS+HM achieves better performance than both BS+BSM and BS+DM for almost all standard benchmarks (1.14×). The reason for such performance improvement across the majority of benchmarks is because the channel bits are generated from *multiple* address bits in hashing-based method, making it possible to adapt to a wide range of access patterns within/across different applications. However, due to the nature of the *pseudo-random* hashing function generation, it may not be optimal for a specific set of access patterns.

Our results show by selecting the optimal address mapping for each individual data structure within an application, SDM+BSM+ML can further improve the performance by 1.16× and 1.27× over the baseline BS+DM for 4 and 32 clusters per application respectively. Increasing the number of clusters leads to better speedup. In comparison to K-Means based method (SDM+BSM+ML), the DL-assisted K-Means (SDM+BSM+DL) achieves 1.33× and 1.43× speedup over

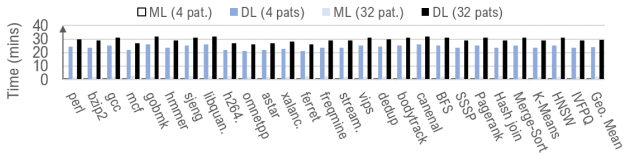

Figure 13: Profiling time for K-Means vs DL-assisted K-Means

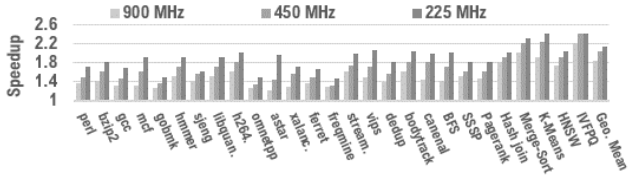

Figure 14: Speedup on CPU when varying the HBM frequency.

the BS+DM for 4 clusters and 32 clusters respectively. Comparing the two methods, we can see DL-assisted K-Means achieves consistently better performance, insensitive to the number of clusters chosen for each application. However, SDM+BSM+ML works well for applications with smaller number of major variables such as perl (1 major variable) and mcf (3 major variables). For applications with a large number of major variables such as omnetpp (65 major variables) and astar (38 major variables), SDM+BSM+ML has nearly no performance improvement (1.04×, 1.09×), but SDM+BSM+DL can deliver a speedup of 1.27× and 1.22×.

For data-intensive benchmarks, the BS+HM achieves similar speedup as that observed in standard benchmarks. In comparison, the and SDM+BSM+DL have higher speedup (1.44× vs. 1.27×, 1.84× vs. 1.41×) on the data-intensive benchmarks than standard benchmarks as they have more memory accesses. The results are consistent with our expectation that emerging data intensive applications would benefit more from our techniques.

We can observe that, even with different input, the proposed method could still detect the access pattern and achieve 1.84× speedup. The reason is that address mapping is a function of data structure, so the changes of the program do not affect the choice of address mapping as long as the data structure and program structure (e.g. memory allocation site) remain the same.

In principle, SDAM achieves higher performance gain with more powerful cores (count, frequency etc.), due to the increased memory requests and thus more chances for memory channel contention. To emulate such effect, we conducted experiments by varying 1) the number of cores, 2) the working frequency of HBM to further stress the memory system. As expected, our measurement results confirmed, on average, the speedup increases from 1.27× to 1.32× when increasing the number of cores from 1 to 4, and the speedup increases by 19% on all the benchmarks when slowing down the HBM to a quarter of its maximum frequency.

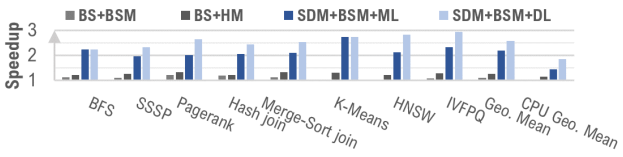

Figure 15: Speedup on accelerators (baseline: BS+DM).

Near-data acceleration: To account for the growing importance of accelerators in future system, we also evaluate the case of offloading a set of data-intensive applications to near memory accelerators. We observe the similar trends as running these benchmarks on CPU in Fig. 15 and show that SDM+BSM+DL outperforms the other four systems for the evaluated benchmark. But they achieves much higher performance gain (2.58× speedup of pairing SDAM with accelerators over the baseline of accelerator without SDAM). It indicates the hardware accelerator can benefit more from our techniques. The reasons are: (i) accelerators typically have more parallelisms, and thus can generate more concurrent memory accesses to external memory; (ii) accelerator tends to have smaller caches, leading to higher cache miss rate.

Profiling time: We measure the profiling time of four SDAM configurations on our workstation with intel i7-9700 CPU for all applications: SDM+BSM+ML (4 patterns), SDM+BSM+ML (32 patterns), SDM+BSM+DL (4 patterns), and SDM+BSM+DL (32 patterns). As shown in Fig. 13, we observe that SDM+BSM+ML (0.3 minutes for 4 patterns and 2 minutes for 32 patterns) has much lower overhead than SDM+BSM+ML (26 minutes for 4 patterns and 29 minutes for 32 patterns). Also, the SDM+BSM+ML approach is more sensitive to the number of clusters (0.3 minutes vs 2 minutes) as the K-Means algorithm converges with fewer iterations for a smaller number of clusters. For benchmarks such as perl and bzip, SDM+BSM+ML (4 patterns) is sufficient to achieve the same speedup as the other three configurations but only takes 0.3 minutes which is much lower than the execution time of the application itself (e.g., 5.9 minutes for perl and 4.4 minutes for bzip). We see similar low overhead for the other benchmark applications when using the appropriate profiling method.

8 RELATED WORK

Address mapping schemes: DRAM address mapping schemes have been previously proposed for single-core CPUs [15, 50], multi-core CPUs [27], and GPUs [11, 30]. Bit-shuffle rearranges the order of the address bits that map to the bank, channel, and column address fields. For instance, Kaseridis et al. [27] use the LSBs as the bank and channel bits to better exploit BLP in the streaming applications. This mapping method is also widely used in commercial computer systems. A common issue of these works is that a global fixed address mapping is configured at the system boot time and cannot be changed at runtime to adapt to different access patterns. Alternatively, hashing-based methods take XOR of several address bits to increase the entropy of the bank address bits to improve BLP on CPU [30, 50] and GPU [30], which could be extended to improve CLP. However, this one-size-fit-all approach relies on the pseudo-random permutation generation that avoids very bad channel contention but cannot achieve optimal CLP for a access pattern [39].

Numerous research work target ways of remapping address dynamically at run-time to improve memory access performance. The most similar works to ours are Akin et. al. [1] and Mohsen et al. [17]. Akin et al. [1] proposes to shuffle the order of physical address, while Mohsen et al. [17] proposes to take the XOR of bank addresses and row addresses with the bank index. However, the common problem of these works is that they select address mapping based on the physical address only whereas the physical address does not contain useful information of the program, such as thread ID, data structure. Without such knowledge, it is difficult to infer the program's behavior accurately at runtime from the statistics collected from hardware to capture the different access patterns in the memory access stream, especially with the interference of other co-running applications. Compared to these hardware-only approaches, our work is a collaborative software/hardware technique. Our SDAM associates the physical address with the high-level program's access semantics and thus can effectively adapt to the different access patterns of applications.

Software-based data-placement optimization: Another lines of research target to leverage the benefits of static program analysis [8, 9, 14, 29, 38, 41, 42], dynamic profiling [42, 44] or a combination of them to guide data-placement in the physical memory hierarchy for performance optimization. The annotations are used to guide the data placement in the different locations of cache [8, 9, 14], heterogeneous memories [12, 13, 26, 32, 35, 42] and NUMA nodes [28, 42] to better exploit MLP and/or to improve the locality. However, a common problem of these methods is that they often rely on a given memory access pattern and become ineffective to cases when applications have different patterns [8, 9, 14, 28, 32, 41, 42]. Moreover, these works can only control the data-placement at coarse-grained granularity [8, 9, 13, 14, 29, 41, 42]. However, as discussed in Section 3, to best exploit CLP in 3D-stacking memory, we need to have more fine-grained control at the granularity of cache block. As discussed in [1, 40], simply decreasing the granularity leads to prohibitively high overhead in both storage and performance, especially for programs with a large number of data structures [41]. In comparison, by clustering data with the same access pattern into chunks, our work can effectively reduce management overhead.

9 CONCLUSION

In this work, we propose software-defined address mapping (SDAM) which is a new system mechanism (architecture/OS support) that enables user program to obtain a direct control of the low-level memory hardware in a more intelligent and fine-grained manner. As a case study, we apply SDAM to 3D-stacking memory to better exploit its CLP by controlling the address mapping, adaptive to the data access pattern to different data structures across applications at runtime, which, is not currently supported in existing systems. We apply machine learning for system optimization to further reduce the overhead of SDAM and improve performance. Our evaluation results not only confirmed the effectiveness of SDAM but also showed new trends with emerging data-intensive applications and new system components such as near memory accelerators will benefit more from the proposed technique. We believe SDAM is an essential step towards Software-Defined Memory (SDM).

REFERENCES

[1] Berkin Akin, Franz Franchetti, and James C Hoe. 2016. Data reorganization in memory using 3D-stacked DRAM. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 131–143.

[2] JEDEC Solid State Technology Association et al. 2012. JEDEC Standard: DDR4 SDRAM. *JESD79-4, Sep* (2012).

[3] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. https://doi.org/10.1145/3373376.3378498

[4] Pierre Baldi. 2011. Autoencoders, Unsupervised Learning and Deep Architectures. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27* (Washington, USA) *(UTLW'11)*. JMLR.org, 37–50.

[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[7] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 117–130.

[8] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. 1996. Compiler-directed page coloring for multiprocessors. In *ACM SIGPLAN Notices*, Vol. 31. ACM, 244–255.

[9] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious data placement. In *ACM SIGPLAN Notices*, Vol. 33. ACM, 139–149.

[10] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David Patterson, and Krste Asanovic. [n.d.]. BOOM v2. ([n. d.]).

[11] Niladrish Chatterjee, Mike O'Connor, Gabriel H Loh, Nuwan Jayasena, and Rajeev Balasubramonian. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 128–139.

[12] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM, 13.

[13] AamerJaleel ChiachenChou and MoinuddinK Qureshi. 2015. BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems. (2015).

[14] Trishul M Chilimbi, Mark D Hill, and James R Larus. 1999. Cache-conscious structure layout. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 1–12.

[15] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P Jouppi. 2010. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 1–11.

[16] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[17] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. 2016. Dream: Dynamic re-arrangement of address mapping to improve the performance of DRAMs. In *Proceedings of the Second International Symposium on Memory Systems*. ACM, 362–373.

[18] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *International Conference on Machine Learning*. 1919–1928.

[19] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006).

[20] Marius Hillenbrand. 2017. Physical Address Decoding in Intel Xeon v3/v4 CPUs: A Supplemental Datasheet. *Karlsruhe Institute of Technology, Tech. Rep.* (2017).

[21] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 78–88. https://doi.org/10.1109/PACT.2011.14

[22] Intel. 2016. Intel Xeon Processor E7 v4 Product Family Datasheet. (2016).

[23] JEDEC. 2013. High bandwidth memory (hbm) dram. *JESD235* (2013).

[24] Xu Ji, Chao Wang, Nosayba El-Sayed, Xiaosong Ma, Youngjae Kim, Sudharshan S Vazhkudai, Wei Xue, and Daniel Sanchez. 2017. Understanding object-level memory access patterns across the spectrum. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[26] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS—OS design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 521–534.

[27] Dimitris Kaseridis, Jeffrey Stuecheli, Jian Chen, and Lizy K John. 2010. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–11.

[28] Christoph Lameter et al. 2013. NUMA (Non-Uniform Memory Access): An Overview. *Acm queue* 11, 7 (2013), 40.

[29] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. 2012. A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 367–376.

[30] Yuxi Liu, Xia Zhao, Magnus Jahre, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Lieven Eeckhout. 2018. Get out of the valley: power-efficient address mapping for GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 166–179.

[31] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[32] Mitesh R Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H Loh. 2015. 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, 126–136.

[33] Erxue Min, Xifeng Guo, Qiang Liu, Gen Zhang, Jianjing Cui, and Jun Long. 2018. A survey of clustering with deep learning: From the perspective of network architecture. *IEEE Access* 6 (2018), 39501–39514.

[34] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. (2010).

[35] Aditya Narayan, Tiansheng Zhang, Shaizeen Aga, Satish Narayanasamy, and Ayse Coskun. 2018. MOCA: Memory object classification and allocation in heterogeneous memory systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 326–335.

[36] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W Keckler, and William J Dally. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 41–54.

[37] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. IEEE, 1–24.

[38] Sujay Phadke and Satish Narayanasamy. 2011. MLP aware heterogeneous memory system. In *2011 Design, Automation & Test in Europe*. IEEE, 1–6.

[39] B Ramakrishna Rau. 1991. Pseudo-randomly interleaved memory. In *Proceedings of the 18th annual international symposium on Computer architecture*. 74–83.

[40] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. 2010. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *ACM Sigplan Notices* 45, 3 (2010), 219–230.

[41] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B. Gibbons, and Onur Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 829–842. https://doi.org/10.1109/ISCA.2018.00074

[42] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B Gibbons, and Onur Mutlu. 2018. A case for richer cross-layer abstractions: Bridging the semantic gap with expressive memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 207–220.

[43] J.L. Wolf, D.M. Dias, and P.S. Yu. 1993. A parallel sort merge join algorithm for managing data skew. *IEEE Transactions on Parallel and Distributed Systems* 4, 1 (1993), 70–86. https://doi.org/10.1109/71.205654

[44] Qiang Wu, Artem Pyatakov, Alexey Spiridonov, Easwaran Raman, Douglas W Clark, and David I August. 2004. Exposing memory access regularities using object-relative memory profiling. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 315.

[45] Xilinx. 2019. AXI High Bandwidth Memory Controller v1.0. (2019).

[46] Xilinx. 2019. UltraScale+ FPGA Product Tables and Product Selection Guide. (2019).

[47] Jialiang Zhang, Soroosh Khoram, and Jing Li. 2017. Boosting the Performance of FPGA-based Graph Processor Using Hybrid Memory Cube: A Case for Breadth First Search. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '17)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/3020078.3021737

[48] Jialiang Zhang, Yang Liu, Gaurav Jain, Yue Zha, Jonathan Ta, and Jing Li. 2019. MEG: A RISCV-Based System Simulation Infrastructure for Exploring Memory Optimization Using FPGAs and Hybrid Memory Cube. In *2019 IEEE 27th Annual*

International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 145–153.

[49] Lixin Zhang, Zhen Fang, Mike Parker, Binu K Mathew, Lambert Schaelicke, John B Carter, Wilson C Hsieh, and Sally A McKee. 2001. The impulse memory controller. *IEEE Trans. Comput.* 50, 11 (2001), 1117–1132.

[50] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*. IEEE, 32–41.