HAM: Hotspot-Aware Manager for Improving Communications With 3D-Stacked Memory

Xi Wang, Antonino Tumeo, *Senior Member, IEEE*, John D. Leidel, Jie Li, and Yong Chen

Abstract—Emerging High-Performance Computing (HPC) workloads, such as graph analytics, machine learning, and big data science, are data-intensive. Data-intensive workloads usually present fine-grained memory accesses with limited or no data locality, and thus incur frequent cache misses and low utilization of memory bandwidth. 3D-stacked memory devices such as Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM) can provide significantly higher bandwidth than conventional memory modules. However, the traditional interfaces and optimization methods for JEDEC DDR devices do not allow to fully exploit the potential performance of 3D-stacked memory with the massive amount of irregular memory accesses of data-intensive applications. In this article, we propose a novel Hotspot-Aware Manager (HAM) infrastructure for 3D-stacked memory devices capable of optimizing memory access streams via request aggregation, hotspot detection, and in-memory prefetching. We present the HAM design and implementation, and simulate it on a system using RISC-V embedded cores with attached HMC devices. We extensively evaluate HAM with over 12 benchmarks and applications representing diverse irregular memory access patterns. The results show that, on average, HAM reduces redundant requests by 37.51 percent and increases the prefetch buffer hit rate by 4.2 times, compared to a baseline streaming prefetcher. On the selected benchmark set, HAM provides performance gains of 21.81 percent in average (up to 34.28 percent), and power savings of 35.07 percent over a standard 3D-stacked memory.

Index Terms—Memory hotspot, 3D-stacked memory, coalescing, prefetching, HBM, HMC, communications

1 Introduction

Emerging High-Performance Computing (HPC) applications, such as graph analytics, machine learning, and big data science are data-intensive. They are memory and latency bound [1], [2], and often exhibit *irregular* behaviors. They use large data sets, stored in pointer-based data structures (graphs, imbalanced trees, unstructured grids, sparse matrices) that lead to fine-grained (word-size) and unpredictable memory accesses. On conventional high-performance processors with complex memory subsystems and multi-level data caches, the non-deterministic memory accesses and poor locality lead to low reuse and high cache miss rates, which inevitably translate to high access latency [3] and poor bandwidth utilization.

3D-stacked memory devices, such as High Bandwidth Memory (HBM) [4] and Hybrid Memory Cube (HMC) [5], provide significantly higher bandwidth with respect to conventional Double Data Rate synchronous Dynamic Random Access Memory (DDR DRAM), and offer an opportunity to better address requirements of data-intensive applications.

- *Xi Wang, Jie Li, and Yong Chen are with the Department of Computer Science, Texas Tech University, Lubbock, TX 79415 USA. E-mail: {xi.wang, jie.li, yong.chen}@ttu.edu.*
- *Antonino Tumeo is with the High Performance Computing Group of Pacific Northwest National Laboratory, Richland, WA 99352 USA. E-mail: Antonino.Tumeo@pnnl.gov.*
- *John D. Leidel is with the Tactical Computing Laboratories, Muenster, TX 76252 USA. E-mail: jleidel@tactcomplabs.com.*

Manuscript received 20 Aug. 2020; revised 2 Mar. 2021; accepted 7 Mar. 2021. Date of publication 18 Mar. 2021; date of current version 17 May 2021.
(Corresponding author: Xi Wang.)
Recommended for acceptance by L. Chen and Z. Lu.
Digital Object Identifier no. 10.1109/TC.2021.3066982

In these devices, the DRAM dies are stacked on top of a logic die via 3D packaging. The logic layer implements the memory controller that manages communication between stacked DRAMs and processors. Well-known commercial devices using this technology include the latest generations of NVIDIA's Graphic Processing Units (GPUs), Intel's Xeon Phi processors, and Fujitsu PrimeHPC FX100. As shown in Fig. 1, Through Silicon Vias (TSV) connect the various layers of the stack, providing high bandwidth for data transactions within the 3D-stacked memory [5], [6]. Parallel high-speed links in the interposer layer connect the logic die to the host processor.

Several studies have started looking at opportunities to embed processing elements in the logic die of 3D-stacked memory, with the objective to overcome the issues brought by the so-called *memory wall* [7]. Near data processing (NDP) and processing in-memory (PIM) [8], [9] aim at reducing the redundant data traffic and overall memory access latency by effectively moving computation to data. However, the amount of logic actually embeddable in the logic layer of a 3D-stacked memory device is limited, both by production (the manufacturing processes for memory devices are not as refined as those used for high-performance cores) and by thermal issues. For these reasons, the logic die is typically only used to either accelerate some specific instructions or to improve the actual memory management operations [9], [10].

From a practical point of view, because of the constraints on the actual logic implementable in a memory controller die and the poor effectiveness of cases, the problem of optimizing memory accesses for data intensive applications in systems with 3D-stacked memory devices becomes a problem of optimizing the communication between a processor

0018-9340 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

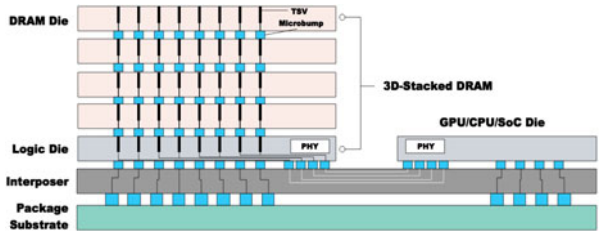


Fig. 1. Example of 3D-stacked memory layout.

(cores and "uncore") and the actual 3D stacked memory device, and in the 3D-stacked memory device itself. Techniques may be applied at the processor, at the memory side, or both, and the overall key result is an improvement of the communication between the two components (reduction in overheads, reduction in data movements, higher bandwidth utilization, and related energy benefits).

From the memory side perspective, a number of efforts have exploited the stacked memory logic layer to implement prefetching [11], [12]. Prefetching is a well-known latency reduction approach adopted in cache-based architectures. The hardware predicts the data that an application will load, typically considering the previous access patterns, fetching them in advance into caches and thus significantly reducing the latency of subsequent communications between memory devices and processors [13], [14]. Conventional prefetchers are, however, optimized for regular workloads with predictable data accesses, such as sequential or unit-stride access patterns. These are not applicable to irregular applications [15], [16]. For instance, neither hardware predictors nor compilers can predict the target address of the scatter operation `a[b[i]] = c[i]` with random indexes `b[i]`.

Another challenge for many data-intensive applications is the frequent generation of memory hotspots, due to the fine-grained nature of their data accesses. *Memory hotspots are frequently accessed memory locations* that may significantly hinder the performance of DRAM devices, due to their banked design. In fact, frequent accesses to the same memory banks increase the probability of bank conflicts, thus increasing the access latency latency [17]. Given the non-deterministic nature of memory accesses in irregular applications, bank-interleaving is insufficient to avoid hotspots.

Additionally, neither developers nor compilers can accurately infer hotspots statically without analyzing the dynamic, runtime data-access patterns [18], thus making identification of hotspots highly challenging for data-dependent algorithms with changing datasets. Hence, new approaches capable of optimizing irregular memory traffic are needed.

As demonstrated in previous works, implementing the prefetching logic in the stacked memory device, rather than in the processor die, allows to further optimize communication between processor and memory device, which is critically expensive (also in terms of energy) because it happen across different dies (on the same interposer) or even across different chips. Coupling hotspot detection with the prefetching logic also allows improving communication performance between the DRAM dies and the logic die in the 3D Stack.

In this paper, we introduce the *Hotspot-Aware Manager* (HAM), a new near-memory component applicable to different 3D-stacked memories (i.e., HBM, HMC, etc.) aimed at improving performance of data-intensive applications that exhibit irregular memory access patterns. HAM resides in the logic die of the 3D-stacked memory. It implements a coalescing mechanism to aggregate fine-grained memory operations, and a hot bank table that identifies hotspots at the level of DRAM rows and banks. HAM includes a hotspot prefetcher that preloads data of "hot" memory locations into memory-side buffers, reducing latency to access contended locations and improving bandwidth utilization of the 3D-stacked memory. In general, HAM provides significant improvements to communication between processor and 3D-stacked memory devices and communication inside the 3D-stacked memory device itself. We evaluated HAM by simulating it in an architecture based on RISC-V cores [19] with an attached HMC device.

This research study makes five specific key contributions. First, we investigate the performance challenges of the data-intensive workloads that exhibit irregular memory access patterns on 3D-stacked memory. We also analyze the memory request distributions to study the characteristics of diverse data-intensive applications. Second, we present the architecture of the hotspot-aware memory manager (HAM), and describe its integration in the logic layer of general 3D-stacked memory devices. Third, we design a coalescing methodology that aggregates the raw requests hitting the same DRAM row with a coalesced request queue (CAQ) within the 3D-stacked memory to eliminate the latency and power cost of redundant memory accesses. Fourth, we introduce a novel memory hotspot detection mechanism to record the frequently accessed physical memory hotspots with a hot bank table at the level of DRAM rows and banks. We also introduce a hotspot prefetcher to reduce the latency of accessing contended locations and improve bandwidth utilization. Fifth, we provide a comprehensive evaluation of the HAM design from the perspectives of the performance, power efficiency, and bandwidth utilization, respectively, with selected data-intensive applications representing diverse irregular memory patterns.

The remainder of this paper is organized as follows. Section 2 provides background and motivations for this research. Section 3 overviews the HAM architecture, describing its near-memory design and its overall operation principles. Section 4 details the design of each HAM component and provides associated analyses. Section 5 discusses the HAM experimental results. Finally, Section 7 draws the conclusions.

2 Background

This section provides background information on 3D-stacked memory row buffer policies and memory hotspot detection.

2.1 3D-Stacked Memory

3D-stacked DRAMs provide substantially higher bandwidth than typical DDR DRAM modules, potentially satisfying requirements of emerging data-intensive workloads in high-performance scientific computing or high-end enterprise computing. These workloads often exhibit unpredictable patterns of fine-grained accesses, leading to poor spatial or temporal locality. Typical examples are sparse

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

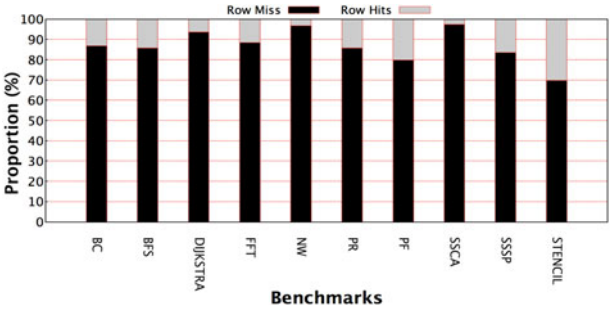


Fig. 2. Row-buffer hit and miss rates.

linear solvers and graph kernels. To better cope with these workloads, and to reduce energy consumption, 3D-stacked memory devices implement smaller row buffers: they respectively are 1 KB large in HBM, and 256 Bytes large in HMC. Recent research work confirms that even smaller rows (e.g., 64 Bytes) provide higher performance improvements and reduced power consumption [20].

However, compared with the 8 KB∼16 KB rows in DDR3, the smaller row size in 3D-stacked memory increases the probability of row buffer misses. This potentially causes more bank conflicts and makes the open-page mode inefficient [21]. Furthermore, the increasingly higher degree of data-level parallelism in applications also reduces the row-buffer hit rates. In Fig. 2 we show the row-buffer hit and miss rates with a variety of data-intensive workloads on a 4 GB HMC device, configured with the open-page mode (environment settings are detailed in Section 5.2). On average, across all the tested workloads, 86.78 percent of the memory accesses generated row-buffer misses, highlighting the limited spatial locality of the data and leading to inefficient communication between host processor and DRAMs.

The open-page policy leaves the DRAM row open after each memory access and caches the data in row buffers, thus a row buffer miss requires writing the data back into the DRAM row, closing the current opened row, and subsequently opening the requested row to complete the memory access. This entire procedure significantly slows down pipelined memory accesses. To reduce power consumption and access latency, the DRAM operation in a 3D-stacked memory device, such as the HMC, follows instead a closed-page policy [5]. In HMC, at the completion of a memory access, the sense amplifiers are precharged and the DRAM row is subsequently closed [22]. Consequently, in a 3D-stacked memory, the row-buffer hit harvesting memory controller [23], which prioritizes requests hitting an opened row, is not applicable for this type of irregular memory accesses.

Furthermore, 3D-stacked memories rely on the vault and channel controllers to eliminate bank conflicts by reordering the request sequences. However, the high number of independent vaults or channels potentially limits the request queue size of each vault or channel [22]. As a result, a limited number of slots in each request queue greatly hinders the effectiveness of request schedulers. Moreover, reordering access streams also introduces additional latency (i.e., read-to-read cycle time (t_{CCD}): 2.5–10 ns) [24] and wastes internal bandwidth on repetitive data transactions (N requests hitting the same row need N transactions). Since banks within the same channel/vault share the data bus, request reordering may also delay the execution of subsequent memory accesses. Additionally, the size of the request queue in each vault or channel is much smaller than the request queue of the Serializer/Deserializer (SerDes) links that connect the 3D-stacked device to the host processor. Because there are multiple internal queues elements (e.g., for link, crossbar, channel/vault, etc.) inside a 3D-stacked memory, delays due to bank conflicts not only stall queues, but also increase power consumption.

2.2 Memory Hotspot

Hotspots in regular workloads are usually mitigated by predicting the access streams and monitoring frequently accessed data segments. Meswani et *al.* [25] have explored the replacement of frequently accessed physical pages between the on/off-package memory devices to reduce the cost of accessing the "hot" physical pages. The approach is ideal for regular workloads. However, irregular applications present large amounts of unpredictable memory accesses that increase cache miss rates and, consequently, the latency of the memory accesses themselves. As a result, such a page replacement approach may not effectively reduce redundant communications to the off-package memory when executing irregular workloads. Because it is complex to predict the access patterns statically through the compiler or dynamically by profiling the processor's performance counters, we investigate the data request patterns from the point of view of the memory. To characterize the request distributions in data-intensive applications, we execute 12 memory-bound workloads with various memory access patterns and then randomly select a time window during runtime to trace the physical destination addresses of the memory accesses.

We cluster the memory traces depending on the values of physical addresses to identify frequently accessed memory regions. Since we cannot predict the number of clusters, we use the unsupervised density-based spatial clustering of applications with noise (DBSCAN) algorithm [26]. We set the epsilon distance of DBSCAN to 1 KB, which is equivalent to the row size of HBM, to group adjacent memory accesses. We set the time windows for the analysis to 10,000 clock cycles. The detailed test environment is described in Section 5.2.

In Fig. 3 circles represent request clusters, and distinct clusters are differentiated by colors. Crosses identify unclustered requests with limited data locality. In each tested workload, we observe multiple request clusters, demonstrating the presence of memory hotspots. We can, thus, conclude that some DRAM banks are more frequently accessed than others. A higher bank utilization leads to a higher probability of bank conflicts [17]. However, if we were able to monitor memory hotspots, we could employ the information to optimize the performance of the memory itself when executing data-intensive applications.

3 Architecture

In this section, we present an overview of the HAM architecture and its operation.

In a 3D-stacked memory, the local controller for each vault or channel resides in the logic layer. HAM is also

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

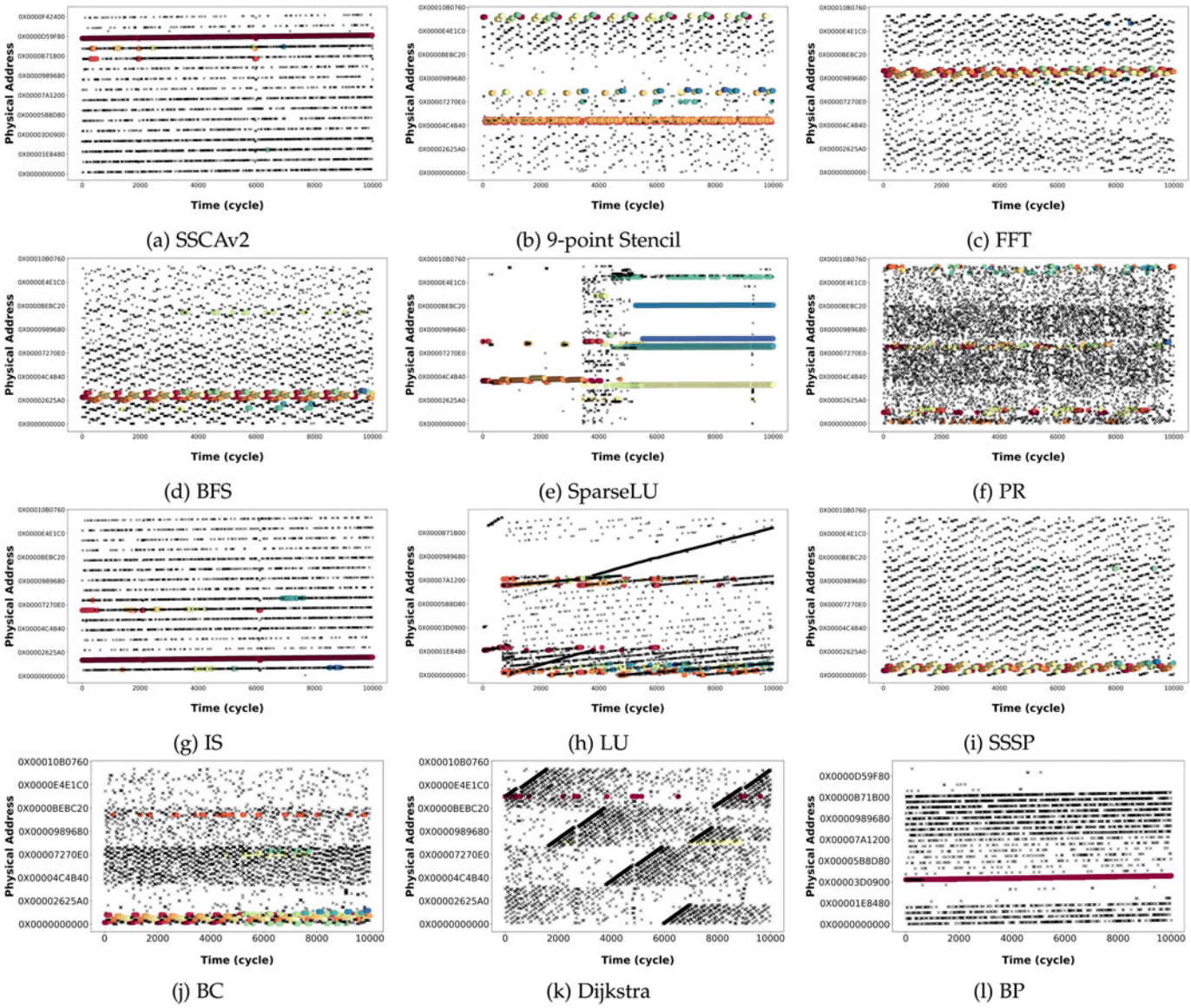


Fig. 3. Memory accesses distribution.

implemented in the memory logic die, providing fine-grained control over bank accesses and optimizing communication between logic and processor dies. High bandwidth memory (HBM) and hybrid memory cube (HMC) are two representative 3D-stacked memory devices, widely utilized in both industry and academia. Hence, to demonstrate HAM's applicability, we illustrate its HAM placement in both these devices, as presented in Fig. 4a. A typical 4 GB HBM device with 4 DRAM dies implements 2 channels per DRAM die, 8 in total. Each channel owns 16 banks that share the same data bus and implements a private controller that manages accesses independently from the other channels [4]. Therefore, as shown on top of Fig. 4a, we implement a HAM unit inside each HBM channel controller. Since each vault/channel in a 3D-stacked memory has private data paths and is independently managed, sharing HAM and prefetch buffers between banks interleaved in distinct vaults/channels would break their inherent parallelism and cause performance degradation. Thus, distributed HAMs and prefetch buffers are preferred.

In the case of HMC, the stacked DRAMs are vertically partitioned into 32 vaults and every 8 vaults are grouped in a quadrant [5]. As shown at the bottom of Fig. 4a, since in HMC each vault within a quadrant shares the same link between the memory device and the host processor, we implement a quadrant-based HAM rather than a vault-based design to minimize the space overhead and die area occupation.

As illustrated in Fig. 4b, HAM consists of four major components: *Coalesced Access Queue*, *hot bank Table*, *Hotspot Prefetcher*, and *Prefetch Buffer*. We briefly introduce each component below, and provide further details in the next section.

3.1 Coalesced Access Queue

The coalesced access queue (CAQ) is a First In First Out (FIFO) queue that aggregates raw requests from the host processor based on the respective row addresses. Each CAQ entry merges the requests hitting the same DRAM

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

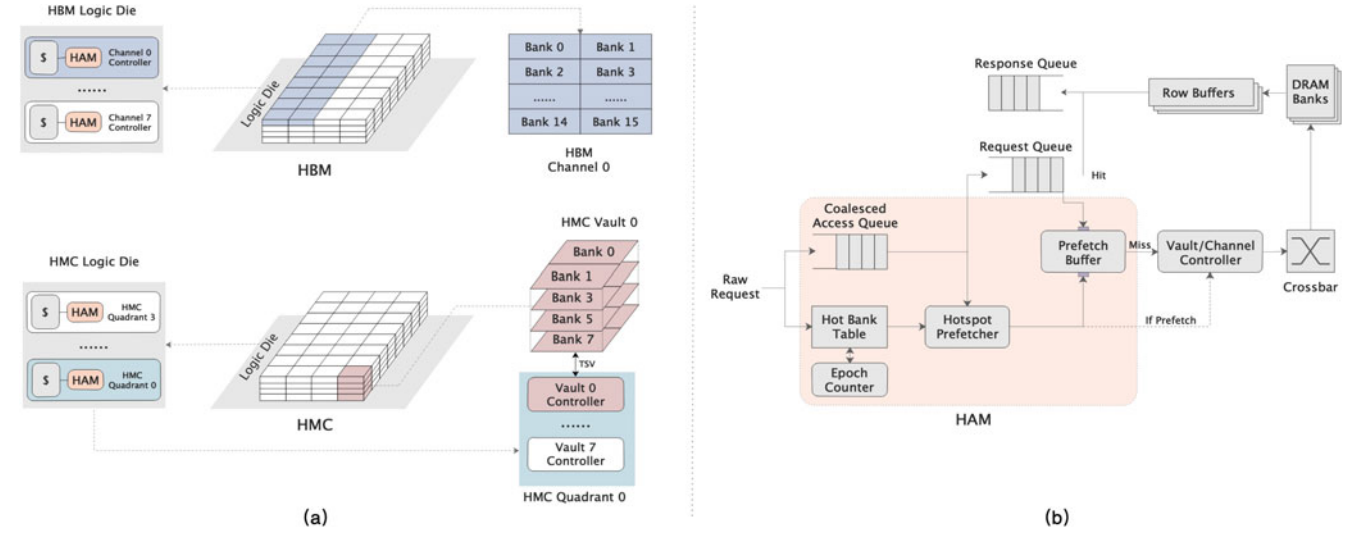


Fig. 4. Architecture of HAM (hotspot-aware manager) in 3D stacked memory.

row and signals the prefetcher when popping out the requests. The aggregated requests are further analyzed by the hotspot prefetcher to enable prefetching, if needed.

3.2 Hot Bank Table

The hot bank table (HBT) records the number of accesses to each DRAM bank in the 3D-stacked memory and identifies whether a requested bank is hot or cold. The HBT analyzes the raw requests and accordingly updates the access counter for each bank and the associated bank status (hot/cold).

As shown in Fig. 4b, each HBT also implements an epoch counter. This counter, which records the total number of accesses, allows setting a time window for learning memory hotspots. Once the number of accesses reaches a predefined value, an epoch ends and the HBT sends the captured bank status to the hotspot prefetcher. The HBT then resets both its entries and the epoch counter to start a new iteration of hotspot detection.

3.3 Hotspot Prefetcher

The hotspot prefetcher manages the prefetching logic. As soon as a request is received from the CAQ, the prefetcher parses it to identify the target bank. It then retrieves the stored bank status and inquires the prefetch buffer to check whether the requested row is cached or not. If the row is found in the prefetch buffer, then no prefetching is needed. Otherwise, the prefetcher continues checking the target row and bank status of this request. Once a hot row or hot bank is recognized, the prefetcher issues a request to the specific vault or channel and loads the entire row. The prefetched data is saved into the in-memory prefetch buffer, which resides in the logic die, thus not consuming any of the external bandwidth between the 3D-stacked memory and the host processor.

3.4 Prefetch Buffer

As shown in Fig. 4b, in HAM we employ a two-port prefetch buffer. The first port handles the accesses from the request queue. The second port manages inquiries from the hotspot prefetcher, which only searches for addresses of prefetch buffer entries. This implies that prefetcher inquiries never manipulate the data cached in the prefetch buffer. To reduce the space and energy overhead of the replicated hardware comparators for the two ports, we employ hardware hashing functions to organize and manage the prefetch buffer.

3.5 Workflow

As shown in Fig. 4b, the raw requests are first simultaneously routed to the CAQ and HBT. As previously discussed, the CAQ aggregates the requests, and the HBT detects the memory hotsposts. The CAQ and HBT work in parallel, and there is no dependency between each other. At the end of each epoch, the HBT only sends the bank status to the prefetcher, while the CAQ forwards the aggregated requests to the request queue following a FIFO policy. If the CAQ entries indicate the need for prefetching, then the aggregated requests and the respective row status are delivered to the hotspot prefetcher after they pop out of the CAQ. If a miss in the prefetch buffer falls into a hot row/bank, then HAM issues a prefetch request to the target vault or channel to perform the DRAM access. Once the requested row is loaded into the row buffer, the entire row is brought into the prefetch buffer to accelerate the subsequent accesses. For common load and store requests (not prefetching), the access streams in the request queue are routed to the prefetch buffer. If there is a hit, then a response is directly returned. Otherwise, the request is conveyed to the specific DRAM bank depending on the request address. Finally, the request is served and a response is inserted into the response queue heading back to the host processor.

4 HAM Design

The prefetch buffer design is analogous to typical direct-mapped data caches, so in this section we focus on describing the other HAM components: the *Coalesced Access Queue* (CAQ), the *Hot Bank Table* (HBT), and the *Hotspot Prefetcher*. In addition, we also analyze the latency of HAM and of the associated page policy.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

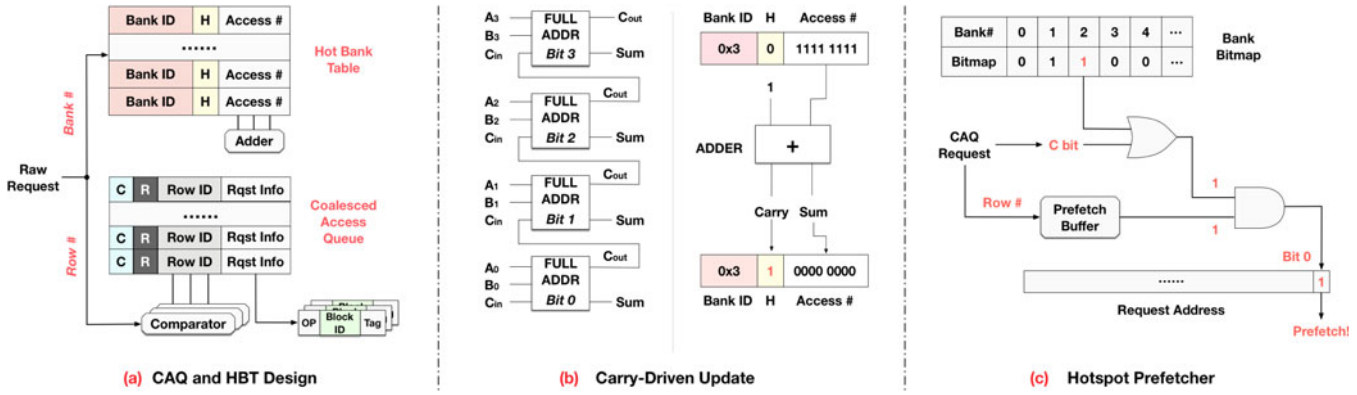


Fig. 5. HAM (Hotspot-aware manager) design.

4.1 Coalesced Access Queue

3D-stacked memory devices have much lower clock rates than processing elements. For example, HBM2 devices have frequencies from 600 to 1000 MHz [27], while modern CPUs easily run from 2 GHz to over 3.3 GHz. We leverage the frequency gap between the processor and the 3D-stacked memory to aggregate and prefetch the request streams. Rather than blocking the input when the request queue is full, the CAQ is designed to aggregate raw requests.

Requests hitting the same row of a specific bank can be served with a single DRAM access, so we set the granularity of request aggregation to the size of the DRAM row buffer. As shown in Fig. 5a, the CAQ aggregates and buffers the raw requests from the host processor. Each CAQ entry consists of four segments: coalescing bit (C), read bit (R), row ID, and request information. The row ID stores the requested DRAM row address, which is utilized as an index for each CAQ entry. The coalescing bit (C) indicates whether two or more read requests hitting the same row are merged into one CAQ entry. The read bit (R) flags the CAQ entry if there exists at least one read request merged in each entry. This bit is checked when requests are popped out. The request information contains the request operation, requested cache block number, and a unique tag associated with the request itself.

Each raw request is first compared with all CAQ entries simultaneously through the hardware comparators. If there is a match, then the raw request is aggregated into a corresponding CAQ entry. Otherwise, the raw request is pushed into the CAQ as a new entry. The R bit is set to 1 if the operation of the inserted raw request is a read. Once a raw request is merged, the respective request information is appended to the target segment. Since requests aggregated in one CAQ entry address the same DRAM row, we only need a few bits to denote the requested cache block. For instance, with 64 Bytes cache lines, each row in HMC (256 Bytes) contains 4 cache blocks. Therefore, only 2 bits are needed to indicate the requested block ID. Once more than one read request are aggregated in the same CAQ entry, then the respective C bit is set to 1, implying the need of prefetching. Otherwise, the C bit remains 0.

According to the miss handling policy of mainstream architectures employing write-back caches, write misses in the last-level cache (LLC) are also translated as read requests to the memory devices. In fact, write operations from the processor store data in a cache line, and the majority of write requests to the memory derives from evictions of dirty cache lines, due to poor data locality. Therefore, there should not be prefetching of data for write requests. We further investigate this phenomenon in Fig. 7. When requests are popped out from the CAQ, the R bit is checked. If the R bit is 0, implying that the requests merged in the CAQ entry are all write requests, then the respective requests are only pushed into the request queue. Otherwise, the requests and associated C bit are also sent to the prefetcher for further analysis.

4.2 Hot Bank Table

Besides the coalescer, we also design a hot bank table to learn the frequently accessed banks in the 3D-stacked memory. The HBT is a lookup table with multiple entries. Each entry corresponds to a specific bank, and the number of entries is determined by the number of banks of a 3D-stacked memory. For instance, a 4 GB HBM device has 16 banks per channel, thus it requires a 16-entry HBT per HBM channel. An HBT entry consists of the bank ID, Hot bit (H), and access counter. The bank ID is utilized as an index for referencing the HBT. The Access Number field serves as a counter that records the number of accesses to each bank. Once a bank entry is referenced in the HBT, the access counter is incremented by one to update the number of accesses to the said bank. If the value stored in the access counter reaches a defined hot-bank threshold, then the hot bit (H) is set to 1. This indicates that the bank is frequently accessed. Otherwise, the H bit remains 0.

4.2.1 *Carry-Driven Update Method*

Since a 3D-stacked memory device handles requests in pipelining, we need to implement a pipelined HBT execution model to hide latency. However, setting the H bit requires an access counter accumulation as well as a comparison between the threshold and the current number of bank accesses. For each bank entry, this procedure induces a pipeline delay and the space overhead of a dedicated hardware comparator. To accelerate the H bit updating logic and avoid comparisons between the access counter and the threshold, the HBT implements a carry-driven update method via a hardware adder shared by all access counters.

As shown on the left of Fig. 5a, a 4-bit adder consists of four full adders. Each adds two bits (i.e., A_0 and B_0, A_1 and

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

B_1, etc.) with the carry bit from the previous full adder. Larger adders are trivially obtained by cascading more full adders. Since a n-bit binary number represents values from 0 to $2^n - 1$, if we set the HBT threshold value to 2^n (n is a non-negative integer) and enable n full adders to complete the n-bit addition, then we can leverage the final output carry bit to indicate the hot banks.

For instance, if the threshold is 256 and the number of accesses to bank 0x3 is 255, then the threshold will be reached at the next access to such a bank, as illustrated on the right of Fig. 5b. Since the threshold (256) is equal to 2^8, only 8 bits are needed to perform the add operation. When the next access to the bank 0x3 happens, the output binary sequence of the adder becomes "1 0000 0000". The carry bit (1) is stored into the H bit and the remaining sum bits are written back to the access counter. In this way, bank 0x3 is identified as a hot bank without any redundant comparisons between the threshold and access counter, thus simplifying the logic of the H bit update and minimizing the associated latency. Once a bank is recognized as a hotspot (i.e., H bit = 1), the accumulation of the associated access counter will be temporarily suspended to save energy until the next epoch.

4.3 Hotspot Prefetcher

4.3.1 Bank Bitmap

Since, at each epoch, H bits of the HBT are reset, we designed a bank bitmap data structure to buffer the bank status. As shown in Fig. 5c, each bank ID corresponds to a single bit in the bank bitmap. 1 and 0 respectively stand for hot and cold. The prefetcher first takes the requested bank as the index to obtain the corresponding H bit in the bank bitmap. The H bit value is then routed to a logic OR gate, together with the C bit of an aggregated request. Meantime, the prefetcher inquires whether the requested row is cached in the prefetch buffer. As discussed in Section 3.4, since the prefetcher employs hardware hashing functions to check the address of a prefetch buffer entry without touching the buffered data, the inquiries from the prefetcher will not delay normal accesses to the prefetch buffer. Return values 0 and 1 stand for hit and miss, respectively. Subsequently, the OR gate output becomes the input of a logic AND gate with the return value from the prefetch buffer. An AND gate output of 1 will enable prefetching.

4.3.2 Prefetch Bit

We also introduce a prefetch bit to hold the AND gate output and help the vault/channel controller to distinguish prefetch requests from common load/store operations. To minimize space overhead, we add a prefetch bit in the physical address of the request itself to identify the requests for prefetching. Since the minimum granularity of a request in HMC and HBM are 16 Bytes and 32 Bytes, respectively, several least significant bits of the request address are actually ignored (4 bits in HMC and 5 bits in HBM). Therefore, we define bit 0 of the physical address as the prefetch bit. In this way, the hotspot prefetcher simply writes the prefetch bit into the request address to trigger the prefetching. Since cache lines are 64 Bytes in the majority of modern architectures, the addresses of common read or write requests from the last level cache are supposed to be 64 Bytes-aligned. Hence, the unused 0∼6 address bits will not collide with the defined prefetch bit.

Given the aforementioned design and configuration, the output of the AND gate is written into the prefetching bit (bit 0) of the request address. If the prefetch bit is set, then the address decoder in each vault/channel can recognize the prefetch request and replace the size of memory operations with the row-buffer size. Once the bank access completes, the entire row is fetched into the prefetch buffer.

Fig. 5c illustrates a prefetcher example. Suppose a request from the CAQ targets bank 2. Thus, the H bit of bank 2 is delivered to the OR gate. If bank 2 is recognized as a hot bank (`bitmap[2] = 1`), then the OR gate outputs 1, regardless of the C bit value of the read request from the CAQ. If the requested row is not found in the prefetch buffer, then the two input bits of the AND gate are both 1. Thus, the AND gate sets the prefetch bit in the request address to 1, and a prefetch request is dispatched.

The HAM design also supports atomic operations. On the CPU side, if an atomic operation does not hit the multi-level data caches, then the memory request size is still identical to the cache line size. Hence, this will not affect the logic of the *Prefetch Indicator*. On the memory side, the 8B∼16B in-memory atomic operations supported in HMC also ignore the address bits 0∼2 as well.

With the standard HMC controller, we can leverage one of the unused bits in the request packet header, such as the bits 58∼60. In HBM, however, the implementation may vary, due to different protocols in memory controllers from different vendors (Open-Silicon, Xilinx, etc.).

4.4 Latency Analysis

We employ a pipelined execution model to hide HAM latency. As discussed in Section 4.2.1, we avoid redundant comparisons and overlap the access counter accumulation with the H bit updates through the carry-driven method. The HBT pipeline implements two stages: bank entry lookup, and H bit update. Similarly, the CAQ operates with a 3-stage pipeline, which includes the row number comparisons, the request merging, and the C/R bits update. Considering the high number of memory accesses produced by data-intensive workloads and the frequency gap between the processor and the 3D-stacked memory, we can easily saturate the request queue and keep the HAM pipeline busy. Since a coalesced request inserted into the request queue is simultaneously dispatched to the prefetcher by the CAQ, we can further hide the latency of prefetching if we have enough request queue entries. Given a default 64-entry request queue and a latency of 2 ns for accessing the prefetch buffer, the aggregated requests from the CAQ need at least 128 ns to approach the prefetch buffer through the pipelined memory access model. In this scenario, considering the average HMC access latency of 93 ns [22], [28], the prefetching latency can also be perfectly hidden, if no bank conflict occurs.

Additionally, by employing a hit-first scheduling policy, each vault controller can still significantly reduce the cost of subsequent accesses hitting the same DRAM row, also in case of a prefetch request delayed by bank conflicts.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

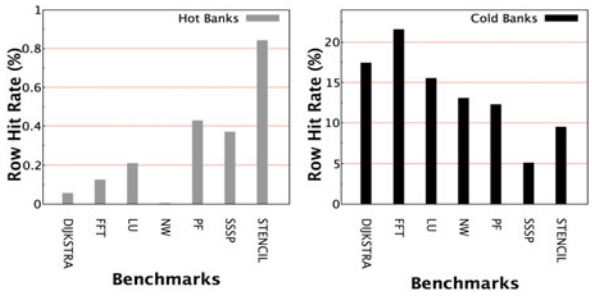


Fig. 6. Row hit rate in hot/cold banks.

4.5 Page Policy

The page policy manages the row buffers of each DRAM bank in the memory device. This directly influences the row hit rate and the cost of row misses. For instance, if a row miss occurs at one bank with the open-page policy, then the latency of writing data back to the currently opened row (if write requests are performed), precharging the current row, and activating a new row, will induce longer delay for the following row accesses, when compared with the closed-page policy that automatically precharges a row after each bank access. Therefore, to enable HAM, we need to design an appropriate page policy for the 3D-stacked memory.

Driven by this motivation, we tested the open-page policy and measured the average row hit rate in both hot and cold banks in HMC. As shown in Fig. 6, on average, only 0.2911 percent of the requests hit rows in hotspots. This implies that HAM fetched the majority of frequently accessed rows into the prefetch buffer. In this case, the probability of reusing prefetched rows in the DRAM banks becomes extremely low. At the opposite of the case with hot banks, we observe much higher row-hit rates for requests falling into the cold banks across each test suite. Nevertheless, row misses (86.51 percent) still dominate accesses to cold memory spots for each data intensive workload in out benchmark set.

Basing on these observations, we employ the *closed-page policy* with HAM to eliminate the overhead induced by row buffer misses. As discussed in Section 2.1, the standard HMC configuration also employs a closed-page policy.

5 Experimental Evaluation

5.1 Simulation Infrastructure

We implemented and simulated the HAM infrastructure on embedded RISC-V cores with the RISC-V RV64IMAFDC instruction set. We first extended the RISC-V *Spike* simulator to trace raw memory requests from multiple cores and to record the latency of the execution pipeline as well as of the cache hierarchy. We then implemented the pipelined HAM design and connected it with the *Spike* simulator to process raw requests along the following steps. First, the row number of each raw request is compared with all CAQ entries to perform request aggregation. Then, the CAQ forwards the aggregated requests to the request queue and the hotspot prefetcher, which issues the prefetch requests if needed. Our simulation infrastructure then routes the coalesced requests to a cycle-accurate 3D-stacked memory simulator, *HMC-Sim 3.0* [29], to obtain data/responses from the 3D-stacked memory. This also allows us to gather statistics related to latency and power consumption of the memory system. Finally, the simulation infrastructure serves LLC misses once it receives data/responses from the 3D-stacked memory.

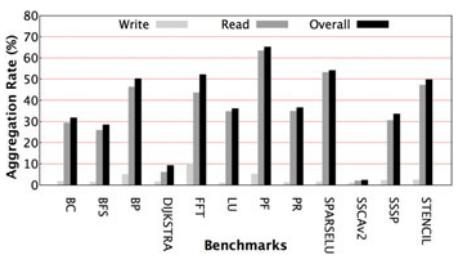


Fig. 7. Aggregation rate (CAQ = 32).

TABLE 1
Simulation Environment Configurations

Parameters	Value
ISA	RV64IMAFDC
Core	8 Cores, 2 GHz
Cache	8-Way, (16KB) L1, (8MB) L2
CAQ Entries	32
Request Queue Entries	64
Threshold	32
Epoch	8192
Prefetch Buffer	Direct-Mapped, 256KB/Quad
HMC	4 Links, 4GB, 256B Row
Avg. HMC Access Latency	93 ns

5.2 Benchmarks and Environment

To evaluate the efficacy of HAM, we selected 12 benchmarks that well represent both dense (aligned) and sparse (random) memory access patterns typical in data-intensive applications, including scientific (physical) simulations, core numerical solvers, and analytics workloads. The benchmark suite includes the Rodinia Benchmarks, Dijkstra Benchmark, Adept Kernel OMP Benchmarks, SSCAv2, Barcelona OpenMP Tasks Suite (BOTS), NAS Parallel Benchmarks (NAS-PB), and GAP Benchmark Suite (GAPBS) [30], [31], [32], [33], [34], [35].

We compiled the aforementioned benchmark set with the RISC-V GCC 7.1 cross compiler and tested it on the RISC-V Linux image based on the extended RISC-V Spike simulator. We set the HBT threshold value at 32, derived from dividing the epoch (8192) by the total number of banks (256) in HMC. This represents the ideal case where memory accesses are evenly distributed to each DRAM bank. We further analyze the impacts of threshold value upon the HBT in Section 5.3.2 to justify our configuration. As discussed in Section 4.5, we employ the *closed-page policy* as the baseline implementation for the 3D-stacked memory. Table 1 shows the detailed parameter set of the simulation environment.

5.3 Results and Analysis

5.3.1 Request Aggregation

To quantify the efficacy of request aggregation, we first define a metric named *aggregation rate* to represent the

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

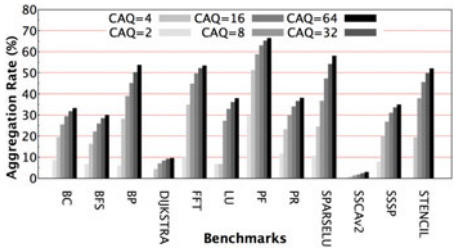


Fig. 8. Aggregation Rate.

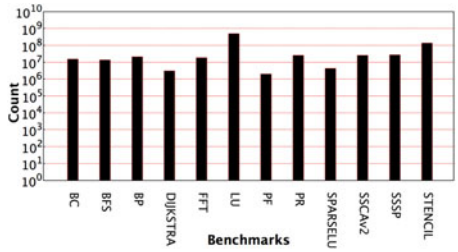


Fig. 9. Reduced request reordering.

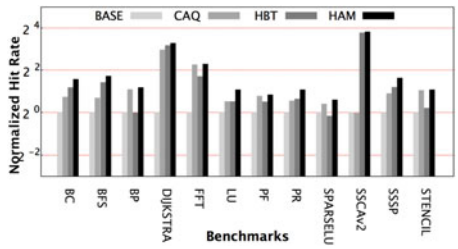


Fig. 10. Prefetch buffer hit rate.

proportion of reduced requests, as derived from:

$$Aggregation\ Rate = \frac{Aggregated\ Rqsts}{Total\ Rqsts}. \quad (1)$$

By measuring the number of total input raw requests and the number of aggregated requests popped out from the CAQ, we derive the aggregation rate of different memory operations for each benchmark. As reported in Fig. 7, the majority of aggregated requests in the CAQ are read requests. These achieve an average aggregation rate of 34.45 percent. Only 3.29 percent of the write requests fall into the same DRAM row. This observation validates the design of our request aggregation model, discussed in Section 4.1. It should be noted that write operations from the CPU directly write to data caches. Write cache misses will issue load operations to the main memory and write operations are completed once the required data is loaded into the cache. As a result, write operations to the main memory are write-back operations from the last-level cache (LLC). Since write-back requests triggered by evicting dirty cache blocks exhibit very limited data locality, optimizations such as prefetching should only be applied to read requests. Overall, HAM aggregates 37.52 percent requests on average via the CAQ. This largely eliminates the redundant memory accesses and the associated power consumption for queuing requests in the 3D-stacked memory.

We also investigate the impact of the CAQ size on the aggregation rate. Fig. 8 shows an increase in aggregation rate, from 8.24 to 40.13 percent, as the number of CAQ entries grows. However, this does not necessarily imply that the CAQ should be as large as possible. Fig. 8 shows that there is a diminishing return as the number of CAQ entries doubles. As the number of CAQ entries increase to 8, 16, 32, and 64, the aggregation rate increases by 43.41, 15.91, 8.64, and 6.96 percent, respectively.

Considering the tradeoff between space overhead and aggregation rate, we select the 32-entry CAQ as the configuration of HAM for the rest of the experimental evaluation.

By combining requests directed to the same DRAM row, request aggregation potentially reduces the frequency of request reordering in each vault controller of the HMC and its associated latency. Thus, we further measure the reduction in request reorderings, and compare it to the baseline HMC with the open-page mode. In both cases, we use a hit-first scheduling policy. As illustrated in Fig. 9, HAM removes in total 0.78 billion of request reorderings, and 65 million on average. Overall, each benchmark presents a considerably lower amount of request reorderings, translating into a reduction of the latency of the memory accesses in each vault.

5.3.2 Prefetching

In this section, we further investigate the efficacy of the hotspot prefetcher. To compare performance, beside HAM (which includes both CAQ and HBT) we further implemented 3 additional prefetching schemes. The baseline scheme (BASE) implements a memory-side streaming prefetcher that loads the entire row (256B) into the prefetch buffer if a miss occurs. The second scheme, which only uses the CAQ (no HBT), prefetches a row if it is requested by two or more read requests in the CAQ. The third scheme, instead, only uses the HBT (no CAQ) to prefetch data residing in the hot banks.

We first record and compare the hit rates of the prefetch buffer for each benchmark, as shown in Fig. 10. HAM exhibits the highest hit rate (61.21 percent) for the prefetch buffer, achieving an average of 4.19X improvement compared with the baseline case. Only using the CAQ or the HBT improves the hit rates by 2.41X and 3.40X, respectively. Notably, the CAQ scheme outperforms the HBT scheme for benchmarks with better data locality, such as BP (Back Propagation), PF (Particle Filter), Stencil, etc. The HBT scheme, instead, provides better performance when executing applications with very limited row locality between adjacent request streams, such as BC (Betweenness Centrality), BFS (Breadth-First Search), SSSP (Single-Source Shortest Paths), etc.

Next, we measure the reduction of redundant prefetch requests in comparison with the BASE scenario to study the prefetching efficiency. As plotted in Fig. 11, CAQ, HBT and HAM reduce prefetch requests in average by 80.81, 68.82, and 61.37 percent, respectively. This observation implies that the HAM issues more prefetch requests as compared to the CAQ and HBT. However, this does not indicate that HAM generates more *redundant* data transactions. Compared with CAQ, HAM provides a prefetch buffer hit rate 18.11 percent higher

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

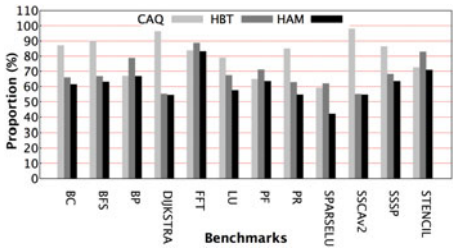


Fig. 11. Prefetching reduction.

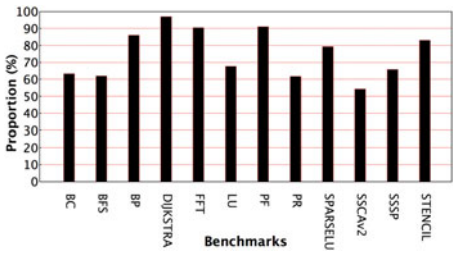


Fig. 12. Row conflict reductions.

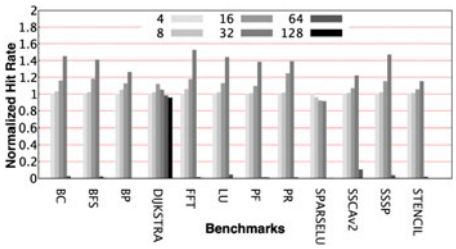


Fig. 13. Prefetch buffer hit rate.

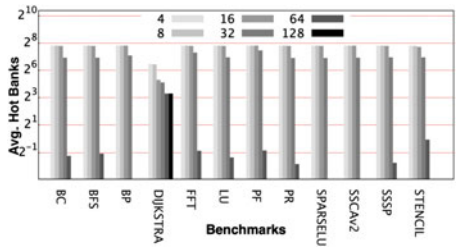


Fig. 14. Avg. hot banks.

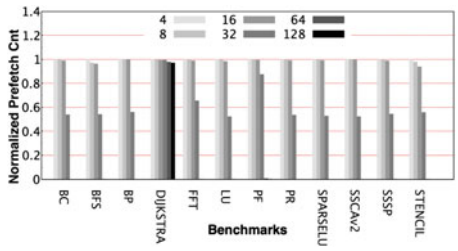


Fig. 15. Prefetch count.

by using 19.44 percent more prefetch requests. This is a reasonable trade-off between performance and overhead.

Although we employ the closed-page policy as the baseline for the evaluation, HAM is additionally capable of optimizing the 3D-stacked memory with an open-page policy. As shown in Fig. 12, we also measure the reduction in row conflicts that HAM provides when using the open-page policy. In comparison with the standard open-page policy, HAM dramatically reduces the row conflicts across each test. DIJKSTRA, FFT (Fast Fourier Transform), and PF shows the highest reduction, with over 90 percent fewer row-conflicts. On average, HAM removes approximately 75.11 percent of the row conflicts, significantly boosting the overall performance by eliminating redundant delays between DRAM accesses.

We also analyze the effects of various thresholds on the HBT. As discussed in Section 4.2.1, the carry-driven update model requires to set the value of the threshold to a power of two. We evaluate the effects of six thresholds from 4 to 128. Since the threshold value only impacts the HBT, we measure the prefetch buffer hit rate only using HBT (the third scheme), removing any performance impacts of CAQ.

We consider the threshold of 4 as the baseline and derive the normalized hit rates as shown in Fig. 13. We can observe that the hit rate increases as the HBT threshold increases from 4 to 32 for the majority of the workloads. On average, the HBT threshold value of 32 provides an improvement of 30.67 percent with respect to the baseline and achieve the peak hit rate. We also observe a rapid decline of the hit rate when the threshold further increases from 32 to 128. This decrease depends on the fact that the larger threshold captures fewer hot banks.

To validate this assumption, we also test each workload with different HBT thresholds and derive the average number of hot banks for each epoch. As presented in Fig. 14, we observe a downward trend from 241.56 to 1.25 of the average number of hot banks per epoch, as the threshold progressively increases from 4 to 128. With a total of 256 banks, a threshold of 4 results in 94.36 percent of the DRAM banks set as hot by the HBT. This behavior can be simply explained with the fact that, if the threshold is too low, then all banks are considered hot and every memory request can then trigger prefetching, resulting in the thrashing of the prefetch buffer. At the opposite, a threshold too high does not allow to identify any potential hot bank, thus not triggering any prefetching. We can also verify this behavior by checking the number of prefetch requests. To this end, we record the number of prefetch requests for each possible threshold value, still considering 4 as the baseline. As shown in Fig. 15, as the threshold increases from 4 to 128, the number of prefetch requests reduces in average by 91.87 percent.

In general, a threshold value of 32 achieves the peak hit rate (30.67 percent higher than the baseline), and reduces the number of prefetch requests by 39.43 percent with respect to the baseline, therefore justifying its choice. As discussed in Section 5.2, this threshold value is derived by dividing the epoch by the total number of banks, considering the ideal case where memory accesses are evenly distributed to each DRAM bank. We expect this heuristic for setting the threshold to hold with longer epochs and/or higher number of banks.

5.3.3 Hotspot Distribution

In this section, we analyze the memory hotspot distribution of the tested applications. First, we randomly select an epoch during the runtime and record the number of

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

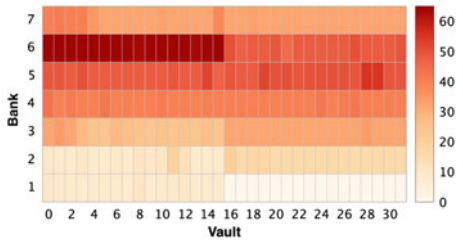


Fig. 16. BC heatmap.

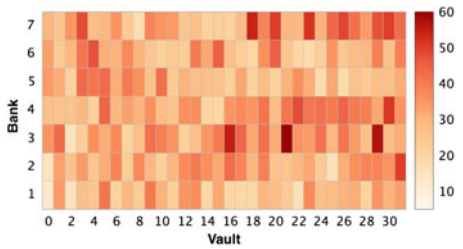


Fig. 17. BFS heatmap.

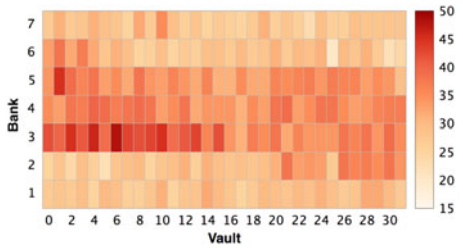


Fig. 18. BP heatmap.

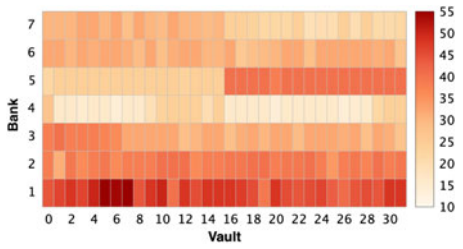


Fig. 19. Dijkstra heatmap.

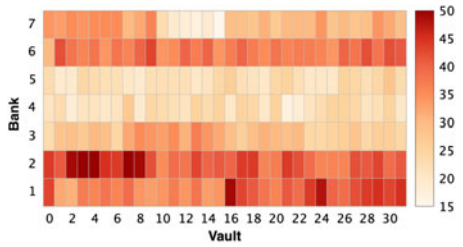


Fig. 20. FFT heatmap.

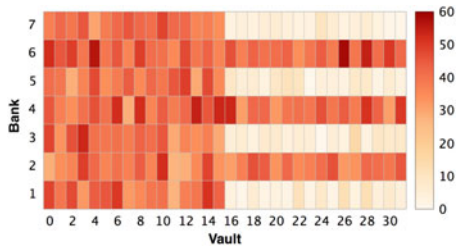


Fig. 21. LU heatmap.

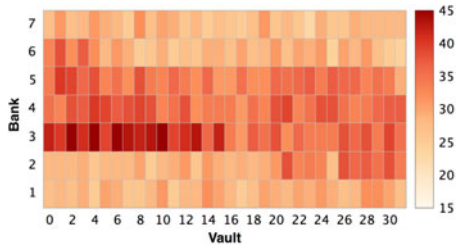


Fig. 22. PF heatmap,

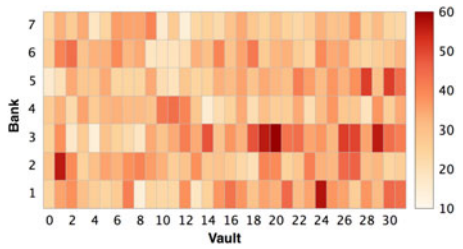


Fig. 23. PR heatmap.

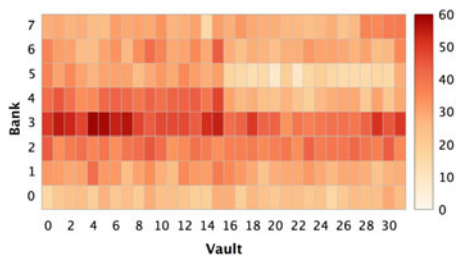


Fig. 24. SparseLU heatmap.

accesses for each bank in the *hot bank table*. To visualize the memory hotspot distribution, we plot the heatmap with a 2D layout of the DRAM banks in HMC for each benchmark as presented in Figs. 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27. In general, we observe distinct hotspots for each tested benchmark within the considered epoch. For example, Fig. 21 reports the memory heatmap of LU, showing that banks 1, 3, 5 and 7 of vaults 16∼32 are rarely accessed. In contrast, the rest of the DRAM banks are more frequently accessed. Overall, for LU, the number of accesses to each bank varies from 1 to 58.

We can also see that for the other benchmarks (such as BFS, PR (Page Rank), SparseLU, SSCA, SSSP and Stencil), some hot DRAM banks are accessed 5 to 6 times more than the cold ones during the selected epoch. These heatmaps confirm that irregular applications may cause significant imbalance in memory accesses.

It should be noted that the memory hotspots are caused by the applications rather than the memory address mapping. Some data regions can be frequently accessed in the program and thus result in memory hotspots. A good memory mapping can alleviate bank conflicts and split frequently accessed data blocks into distinct DRAM rows.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

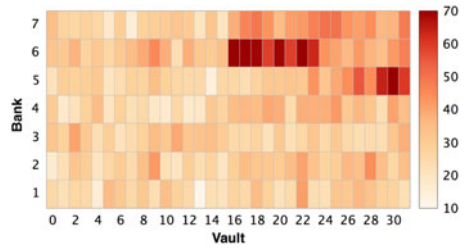


Fig. 25. SSCA heatmap.

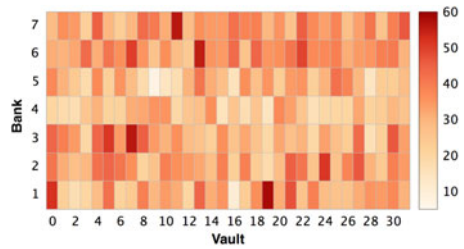


Fig. 26. SSSP heatmap.

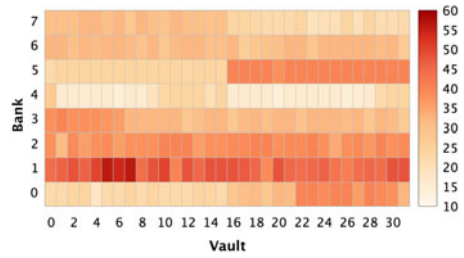


Fig. 27. Stencil heatmap.

However, hotspots cannot be simply eliminated by refining the memory address mapping.

The vault-interleaved data mapping in HMC evenly distributes data along the sequence of bank 0 of vault 0, bank 0 of vault 1, bank 0 of vault 2, etc. This potentially mitigates the formation of hotspots when applications perform data accesses on easily partitionable data structures with regular strides (e.g., dense matrices and tensors). Such vault-interleaving in HMC can help alleviate the performance penalty of hotspots as compared to the bank-interleaving in conventional DDRx devices with large DRAM rows. However, this data mapping cannot efficiently avoid hotspots with the unpredictable, data-dependent and fine-grained memory accesses patterns of irregular applications (e.g., graph kernels, highly sparse matrices) as observed in Figs. 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27, thus requiring more refined hot-spot management approaches like HAM to optimize the overall performance.

5.3.4 Communication

We also investigate if HAM can improve the communication between processor and 3D-stacked memory by analyzing the bandwidth utilization. Each HMC memory access couples a pair of packets (request and response). Each packet contains a header, a payload, and a tail [5]. Header and tail contain control information, such as cyclic redundancy check, error codes, request tag, etc. Every request or response packet has

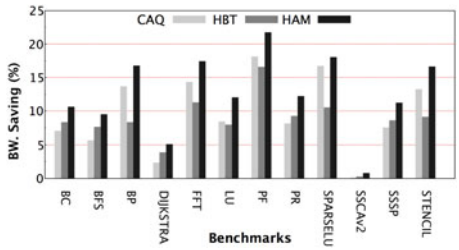


Fig. 28. Bandwidth saving.

16 Bytes of control information, thus a complete memory transaction involves 32 Bytes of control overhead, regardless of the data payload. For instance, an HMC request with 8 Bytes payload requires 40 Bytes of data movement in total, implying that 80 percent of the bandwidth is wasted on transferring the control information. As a result, the large amount of fine-grained raw requests in irregular workloads may lead to a significant proportion of bandwidth wasted on control overheads in the communication between CPU and 3D-stacked memory device. Such overhead in communications inevitably results in performance degradation, including higher memory access latency, and increased power consumption. Since the size of the headers and tails for each request is fixed, issuing larger requests achieves higher bandwidth utilization. Request aggregations and prefetching in HAM can effectively reduce the redundant control overhead for requests hitting the same DRAM row. We first measure the number and size of transactions in a standard HMC device and compare it with the traffic generated when respectively employing CAQ, HBT, and HAM, to determine the bandwidth saving between the host processor and the 3D-stacked memory. As shown in Fig. 28, HAM improves HMC bandwidth utilization by over 10 percent for 9 benchmarks. In Particle Filter (PF), HAM even achieves a bandwidth utilization improvement of 21.77 percent. Overall, CAQ, HBT and HAM reduce redundant transactions in communications with HMC across the tested workloads by 9.61, 8.47 and 12.68 percent, respectively. The measurements confirm that HAM effectively increases bandwidth utilization of the 3D-stacked memory, especially for memory-bound workloads.

Additionally, the 4-link connections employed between HMC and host processor can achieve a peak bandwidth of 320 GB/s by issuing the largest request packets (256B) in HMC [5], [21], [36]. However, since irregular workloads exhibit sparse request distributions featuring limited spatial locality, the majority of aggregated request sizes are 64B and 128B. As a result, the tested workloads achieve an average peak bandwidth of 253.63 GB/s between HMC and CPU.

5.3.5 Space Overhead

As discussed in Section 4.2, each HBT entry consists of bank ID, H bit, and access counter. Since each quadrant of a 4 GB HMC has 64 banks, the bank ID segment occupies 6 bits per HBT entry. Furthermore, we employ a 16-bit access counter supporting up to 64K accesses, which is large enough for a hot bank threshold of 32. Thus, each HBT entry requires a 23-bit buffer, including the H bit. In total, for a 256-bank HMC device, only 0.72 KB are needed. Moving to CAQ, each entry requires: 1 bit for the coalescing (C) bit, 1 bit for

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

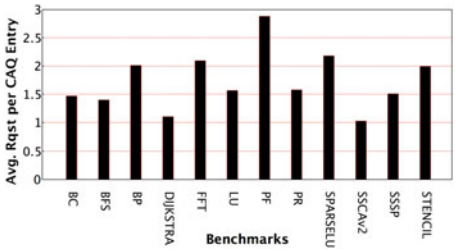


Fig. 29. Avg. request/CAQ-entry.

the read (R) bit, 24 bits for the row ID, and a number of bits to store the request information segments. The latter size depends on the number of aggregated requests per CAQ entry. In order to configure an appropriate size for the request information segment, we measured the average requests merged in each CAQ entry. As shown in Fig. 29, on average each CAQ entry merges 1.74 requests across all our benchmarks. Thus, we configure the request information segment to hold up to 8 requests. Considering that each request needs 20 bits, including a 2-bit operation (read/write/atomic memory operation), a 2-bit block ID (as discussed in Section 4.1) and a 16-bit transaction tag supporting up to 64K requests on the fly, four CAQs (one per quadrant) require the space of 2.91 KB in total. Since each hotspot prefetcher also uses 64 bits for the bank bitmap, the overall buffer space required by HAM is 3.66 KB.

In addition, the space complexity of HAM is proportional to the number of banks. Differently from previous hot page replacement approaches, such as the one in [25] (HPR), which implemented a counter for each TLB entry and page table entry to detect hot pages [25], HAM has a much lower buffer space overhead. Fig. 30 shows that, as the memory size increases from 2 GB to 256 GB, the space overhead of HPR rapidly increases from approximately 1.06 MB to 136 MB. However, the space overhead of HAM only increases from 1.83 KB to 234 KB, i.e., HAM is up to 595.15 times smaller than HPR.

5.3.6 Performance

We measure and report the runtime statistics of the HMC device for each benchmark to quantify the performance improvements on the memory subsystem provided by HAM. This analysis directly summarizes the overall improvements (reduced latency, better bandwidth utilization) provided by HAM for the communication between host processor and 3D-stacked memory, as well as the communication between logic die and DRAM dies in the 3D-stacked memory device itself. First, we evaluate the baseline case with the closed-page HMC device and the standard

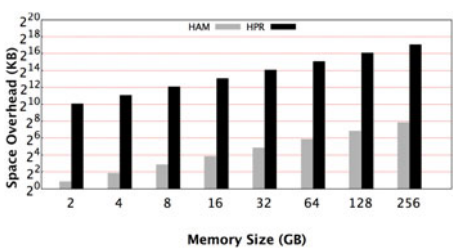


Fig. 30. Space overhead comparison.

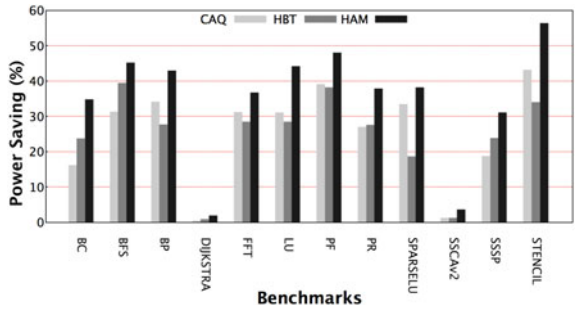


Fig. 31. Power saving.

HMC controller [5]. Then, we test each benchmark with CAQ, HBT, and HAM respectively, to quantify the corresponding performance gain with respect to the baseline. As confirmed by Fig. 32, we observe significant performance improvements for all the memory-bound benchmarks. On average, CAQ, HBT, and HAM provide memory performance improvements of 16.29, 14.51 and 21.81 percent, respectively. The best performance improvements, 31.72 and 32.33 percent, are recorded for PF and Stencil, respectively. These results confirm the effectiveness of the HAM design for 3D-stacked memories.

5.3.7 Power Saving

We investigate the power saving provided by HAM for 3D-stacked memory devices. We first measure the power consumption of a standard HMC device with each benchmark to obtain the baseline costs. Then, we enable CAQ, HBT and HAM, respectively, comparing the measured power consumption to the base case and deriving the power saving. On average, CAQ, HBT and HAM reduce power consumption by 25.57, 24.34 and 35.07 percent, respectively, as observed in Fig. 31. Overall, HAM provides up to 56.41 percent power savings on all the tested benchmarks with irregular behaviors. Improvements in power consumption also are are a direct consequence of the improvements in communication efficiency between host processor and 3D-stacked memory and within the 3D-stacked memory device itself.

6 Related Work

6.1 Aggregation

Request aggregation is a technique that buffers, reorders, and then combines many small requests into a lower number of large requests [37]. Request aggregation is widely utilized in modern single-instruction multiple-data (SIMD)

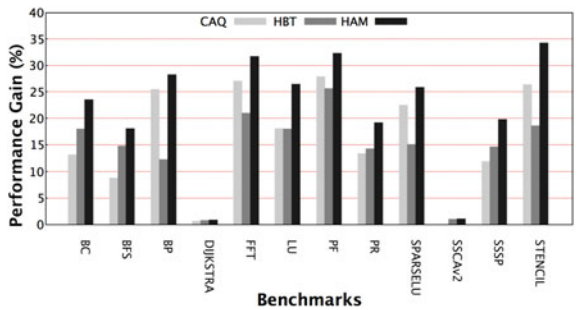


Fig. 32. Performance improvement.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

architectures to improve memory bandwidth utilization [38]. In GPUs, a memory coalescer is responsible for combining memory accesses to the same cache line made by multiple threads in a warp [39]. In addition to aggregated cache accesses, many out-of-order processors employ non-blocking caches and merge data requests for cache misses on the same cache line through the Miss Status Holding Registers (MSHRs) [37], [40].

However, the maximum bandwidth for 3D-stacked memory devices (such as HMC) is achieved with transactions that employ large request packets (e.g., 128B or 256B) rather than small ones (e.g., 32B or 64B) [21], [36]. Cache-line based coalescing methodologies only produce requests at cache-line size, which in current architectures are significantly smaller than the requests in a 3D-stacked memory device, and therefore are not sufficient to reach the peak throughput of such devices [37], [39], [41]. Hence, other coalescing mechanisms, such as the one provided by HAM, are required.

6.2 Prefetching

Prefetching is a widely utilized technique that predicts and fetches the data before it is accessed to reduce memory access latency. Traditional core-side prefetchers residing in the processor preload data into the cache or stream buffers to efficiently serve upcoming memory requests by exploiting a predictor [15], [16]. Existing prefetching techniques such as stride prefetching [42], stream buffers [43], etc., have been proven to improve performance of workloads with regular memory access patterns. There are some efforts exploring prefetching for irregular memory access patterns [18], [44] that leverage the correlation between data accesses. However, core-side prefetching [18], [44] utilizes a significant amount of bandwidth between processor and memory devices. Moreover, mispredicted prefetching causes unnecessary evictions of cache lines or stream buffers, inducing higher latency. Data-intensive applications with irregular memory accesses leave most data loaded in a cache line untouched when the line is evicted [18], leading to "cache-trashing" effects. Cache-trashing, in turn, leads to redundant data transactions between memory and core, wasting memory bandwidth and power.

HAM implements memory-side prefetching, improving utilization of the bandwidth between memory device and processor, and lowering power consumption by removing redundant memory transactions. By placing prefetchers inside the memory devices, the prefetched data remains in the memory controller, and they are only delivered to the core-side cache on demand [45]. Because the memory controller is aware of detailed memory mappings, internal bank states, request scheduling, queuing states, etc., these information can be further exploited to optimize prefetching.

6.3 3D-Stacked Memory

3D-stacked memory provides more opportunities for the optimization of the communication in the memory device itself than traditional DDRx devices. As discussed in Section 2.1, 3D-stacked memory devices (HBM/HMC) employ narrow rows that bring fewer data into the sense amplifiers compared to DDRx devices. This makes prefeching an entire row less wasteful than in a DDRx memory device, and more suitable for the irregular access patterns of data-intensive workloads that may only require a single memory word of the whole row. At the same time, this feature (as previously shown in Fig. 2) provides additional opportunities to optimize the behaviors of a 3D-stacked memory device with irregular workloads.

While the logic die of a 3D stacked memory with current fabrication constraints cannot really implement entire high-performance cores, it can definitely provide enough area and a sufficient thermal envelope to accommodate some additional components as part of the actual memory controller that can further optimize access to the DRAM dies on top and communication with the actual processor die (through links either on the same substrate, or on a conventional printed circuit board).

The Through Silicon Vias (TSVs) among logic die and DRAM dies in the 3D-stacked memory provide bandwidth in abundance, allowing to efficiently perform memory-side prefetching without consuming link bandwidth, which in turn remains available to the processor for other useful memory operations.

However, there is little research exploring prefetching in 3D-stacked memory. Literature [11] proposed a prefetcher in each vault for regular workloads, based on the frequency of row-buffer conflicts in HMC. However, this is not effective when running data-intensive applications with poor row locality. Compared with the quadrant-based prefetching in HAM that serves requests at the link side, private prefetchers and vault-local caches require more power and do not improve performance as much. For instance, requests hitting the prefetch buffer still need to go through multiple queues and require performing redundant data transactions between the links and vaults to access the data from local prefetch buffers. On the other hand, conventional in-memory prefetchers are not applicable to 3D-stacked memories, due to the use of multiple parallel high-speed links between the processor die and the stacked memory device. Employing these methods in a 3D-stacked memory would require synchronization across distinct links, thus jeopardizing the inherent memory-level parallelism.

7 Conclusion

In this work, we have introduced a novel hotspot-aware manager (HAM) infrastructure and the associated methodologies for request aggregation, memory hotspot detection, and prefetching. We designed HAM as near-memory component added in the logic die of a 3D-stacked memory device to solve, in particular, issues caused by data-intensive applications with irregular memory access patterns. While 3d-stacked memories today provide a considerable increase in bandwidth and reduction in latency with respect to conventional DDR devices, irregular memory access patterns lead to a significant under-utilization of their additional capabilities, making the tradeoffs of density, speed, and costs much less appealing than for other workloads. By identifying memory hotspots, HAM allows to efficiently prefetch hot DRAM rows or banks, while at the same time enabling aggregation of requests hitting those rows or banks. This improves performance, power efficiency, and

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

increases bandwidth utilization, both inside the 3d-stacked memory device and between the processor and the memory device itself. Our evaluation shows that with HAM, on a set of representative benchmarks for data-intensive applications, 37.52 percent of the requests are aggregated and 61.21 percent hit the prefetch buffer. On this benchmark set, on average, HAM improves the overall memory system performance by 21.81 percent and reduces the power consumption by 35.07 percent with minimal modifications to a standard 3D-stacked memory device. These improvements are a direct consequence of the optimized communication between host processor and 3D-stacked memory and within the 3D-stacked memory itself enabled by HAM. These results and observations confirm the impact of HAM on the design of custom system architectures for the increasingly important class of data-intensive applications.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant CCF-1409946, Grant CCF-1718336, Grant OAC-1835892, Grant CNS-1817094, and Grant CNS-1939140, and in part by the Pacific Northwest National Laboratory's High Performance Data Analytics (HPDA) Program and Data-Model Convergence (DMC) Initiative.

References

[1] R. Panda and L. K. John, "HALO: A hierarchical memory access locality modeling technique for memory system explorations," in *Proc. Int. Conf. Supercomput.*, 2018, pp. 118–128.

[2] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proc. Int. Symp. Code Gener. Optim.*, 2013, pp. 305–317.

[3] L. Schares *et al.*, "A throughput-optimized optical network for data-intensive computing," *IEEE Micro*, vol. 34, no. 5, pp. 52–63, Sep./Oct. 2014.

[4] "JEDEC standard high bandwidth memory (HBM) DRAM specification," 2013. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd235a

[5] "Hybrid memory cube specification 2.1," 2015. [Online]. Available: https://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf

[6] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus Sside channel," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 94–106.

[7] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, 1995.

[8] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *Proc. Int. Conf. Parallel Archit. Compilation*, 2015, pp. 113–124.

[9] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.

[10] J. Huang, *et al.*, "Active-routing: Compute on the way for near-data processing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 674–686.

[11] M. M. Rafique and Z. Zhu, "Camps: Conflict-aware memory-side prefetching scheme for hybrid memory cube," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 63.

[12] M. Islam, K. M. Kavi, M. Meswani, S. Banerjee, and N. Jayasena, "HBM-resident prefetching for heterogeneous memory system," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2017, pp. 124–136.

[13] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 1–13.

[14] C. Ortega, V. Garcia, M. Moreto, M. Casas, and R. Rusitoru, "Data prefetching on in-order processors," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2018, pp. 322–329.

[15] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *ACM Trans. Archit. Code Optim.*, vol. 10, 2013, Art. no. 2.

[16] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2010, pp. 213–224.

[17] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 24–35.

[18] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 178–190.

[19] X. Yu, C. J. Hughes, N. Satish, and S. Devadas., "IMP: Indirect memory prefetcher," in *Proc. 48th Int. Symp. Microarchit.*, 2015, pp. 178–190.

[20] N. Chatterjee, *et al.*, "Architecting an energy-efficient dram system for GPUs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 73–84.

[21] P. Rosenfeld, "Performance exploration of the hybrid memory cube," Ph.D. dissertation, Dept. Elect. Eng., Univ. Maryland, College Park, MD, 2014.

[22] R. Hadidi, B. Asgari, B. A. Mudassar, S. Mukhopadhyay, S. Yalamanchili, and H. Kim, "Demystifying the characteristics of 3D-stacked memories: A case study for hybrid memory cube," in *Proc. IEEE Int. Symp. Workload Characterization*, 2017, pp. 66–75.

[23] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 128–138.

[24] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "The blacklisting memory scheduler: Balancing performance, fairness and complexity," 2015, *arXiv:1504.00390*.

[25] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *Proc. IEEE 21st Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 126–136.

[26] M. Ester *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discovery Data Mining*, 1996, pp. 226–231.

[27] "High Bandwidth Memory (HBM2) Interface Intel® FPGA IP User Guide," 2018. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-20031.pdf

[28] M. Gokhale, S. Lloyd, and C. Macaraeg, "Hybrid memory cube performance characterization on data-centric workloads," in *Proc. 5th Workshop Irregular Appl., Architectures Algorithms*, 2015, Art. no. 7.

[29] J. D. Leidel and Y. Chen, "HMC-Sim: A simulation framework for hybrid memory cube devices," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1465–1474.

[30] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

[31] U. Brandes, "A faster algorithm for betweenness centrality," *J. Math. Sociol.*, vol. 25, pp. 163–177, 2001.

[32] N. J. *et al.*, "Adept deliverable D2.3 - updated Rreport on adept benchmarks," 2015. [Online]. Available: http://www.adept-project.eu/images/Deliverables/Adept%20D2.3.pdf

[33] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 124–131.

[34] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "NAS parallel benchmark results," in *Proc. ACM/IEEE Conf. Supercomputing*, 1992, pp. 386–393.

[35] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015, *arXiv:1508.03619*.

[36] R. Hadidi *et al.*, "Performance implications of NoCs on 3D-stacked memories: Insights from the hybrid memory cube," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2018, pp. 99–108.

[37] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, Jan. 2011.

[38] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM latency divergence in irregular GPGPU applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 128–139.

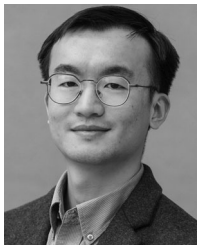
Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.

[39] J. Kloosterman, *et al.*, "Warppool: Sharing requests with inter-warp coalescing for throughput processors," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 433–444.
[40] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proc. 13th Annual. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2015, pp. 12–22.
[41] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing memory access patterns for heterogeneous systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.
[42] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newslett.*, vol. 12, pp. 102–110, 1992.
[43] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Comput. Archit. News*, vol. 18, pp. 364–373, 1990.
[44] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2013, pp. 247–259.
[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 *Proc. 22nd Int. Conf. Parallel Architectures Compilation Techn.*, 2013, pp. 289–298.

Xi Wang received the MS and PhD degrees in computer science from Texas Tech University, in 2016 and 2020, respectively, under the advisement of Dr. Yong Chen and Dr. John D. Leidel. He is a research scientist of the RISC-V International Open Source (RIOS) Laboratory, Tsinghua University. His research interests include computer architecture design, memory systems, data-intensive computing, compilers, machine-learning based system optimizations, and parallel programming models.

Antonino Tumeo (Senior Member, IEEE) received the MS degree in informatic engineering and the PhD degree in computer engineering from Politecnico di Milano in Italy, in 2005 and 2009, respectively. Since February 2011, he has been a research scientist with the PNNL's High Performance Computing group. He Joined PNNL in 2009 as a post doctoral research associate. Previously, he was a post doctoral researcher at Politecnico di Milano. His research interests are modeling and simulation of high performance architectures, hardware-software codesign, FPGA prototyping, and GPGPU computing.

John D. Leidel received the PhD degree in computer science from Texas Tech under the advisement of Dr. Yong Chen. He is the founder and chief scientist of Tactical Computing Laboratories where he leads efforts in programming model, compiler and algorithm research for advanced computing architectures. He is also an adjunct researcher with the Data Intensive Scalable Computing Laboratory at Texas Tech University. His research interests include programming model exploitation of advanced architectures, data intensive computing, domain specific languages for hardware co-design, and advanced compiler optimization techniques.



Jie Li received the MS degree from Texas Tech University, in 2019. He is working toward the PhD degree in computer science at Texas Tech University. His advisor is Dr. Yong Chen in Data-Intensive Scalable Computing Laboratory (DISCL). His current research interests include high-performance computing (resource management & job scheduling, power and Eenergy efficiency, system monitoring) and computer srchitecture.

Yong Chen is an associate professor and director of the Data-Intensive Scalable Computing Laboratory, Computer Science Department, Texas Tech University. He is also the site director of the NSF Cloud and Autonomic Computing Center, Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing. For more information please visit http://www.myweb.ttu.edu/yonchen/.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Texas Tech University. Downloaded on July 01,2021 at 18:00:08 UTC from IEEE Xplore. Restrictions apply.