MAC: Memory Access Coalescer for 3D-Stacked Memory

Xi Wang
Texas Tech University
xi.wang@ttu.edu

Antonino Tumeo
Pacific Northwest National Laboratory
Antonino.Tumeo@pnnl.gov

John D. Leidel
Tactical Computing Labs
jleidel@tactcomplabs.com

Jie Li
Texas Tech University
Jie.Li@ttu.edu

Yong Chen
Texas Tech University
yong.chen@ttu.edu

ABSTRACT

Emerging data-intensive applications, such as graph analytics and data mining, exhibit irregular memory access patterns. Research has shown that with these memory-bound applications, traditional cache-based processor architectures, which exploit locality and regular patterns to mitigate the memory-wall issue, are inefficient. Meantime, novel 3D-stacked memory devices, such as Hybrid Memory Cube (HMC) and High Bandwidth Memory (HBM), promise significant increases in bandwidth that appear extremely appealing for memory-bound applications. However, conventional memory interfaces designed for cache-based architectures and JEDEC DDR devices fit poorly with the 3D-stacked memory, which leads to significant under-utilization of the promised high bandwidth.

As a response to these issues, in this paper we propose MAC (Memory Access Coalescer), a coalescing unit for the 3D-stacked memory. We discuss the design and implementation of MAC, in the context of a custom designed cache-less architecture targeted at data-intensive, irregular applications. Through a custom simulation infrastructure based on the RISC-V toolchain, we show that MAC achieves a coalescing efficiency of 52.85% on average. It improves the performance of the memory system by 60.73% on average for a large set of irregular workloads.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; • **Hardware → Emerging interfaces**; **Memory and dense storage**.

KEYWORDS

Memory Coalescing, 3D-Stacked Memory, Irregular Applications

1 INTRODUCTION

Graph analytics, data mining, machine learning, and data-driven scientific computing are application areas that increasingly need in-memory processing (in contrast to disk storage) to provide the level of performance required to meet strict response time. The datasets accessed in these applications are often unstructured and sparse, and only highly dynamic, multidimensional data structures - such as graphs, trees, and grids - are capable of organizing the collected data in a supportive manner. Unfortunately, these pointer or linked list based data structures, as well as the algorithms used to explore them, determine *irregular* behaviors. While they have a huge amount of latent parallelism and typically are memory bandwidth bound, they perform fine-grained, unpredictable data accesses with no temporal or spatial locality [39], making the complex hierarchical caches of conventional high-performance processor ineffective. In general, memory latency reduction techniques that are based on capturing recurring patterns and predicting future access locations, such as prefetching [19, 29, 35], are not only ineffective, but often detrimental, as they may lead to an increased number of cache misses.

On the other hand, new 3D-stacked memory devices, such as High Bandwidth Memory (HBM) or Hybrid Memory Cube (HMC), can provide a significant increase in bandwidth, at the price of a lower memory density, as demonstrated by the latest generations of NVIDIA's Graphic Processing Units (GPUs) and Intel's Xeon Phi processors. These devices employ 3D packaging to stack Dynamic Random Access Memory (DRAM) dies on top of a logic layer that typically implements the memory controller. Through Silicon Vias (TSV) [3, 5] then connect various layers. In current commercial products, the 3D-stacked memory devices are attached to the main processor dies through an interposer. Exploiting the independent memory banks, organized in interleaved vaults across the stacked dies, an HMC device can provide up to 320 GB/s of data bandwidth [25] to a connected processor, thus potentially satisfying the increasing bandwidth requirements of many data-intensive workloads.

In the context of stacked memory architectures, a number of solutions attempt to improve the performance and utilization by implementing processing elements directly into the logic layer under the DRAM stack, enabling near data processing (NDP) [17, 22] and processing in-memory (PIM) [7, 13, 46] by effectively moving computation to data. While these designs are forward looking, there are still doubts about the size, type, and speed of the processing elements that can currently be implemented in the logic layer. Thus, we typically see these approaches used in scenarios like offloading simple, commonly repeated operations.

Other approaches explore optimizing the memory path by implementing Dynamic Memory Coalescing (DMC) with the objective of minimizing the amount of memory transactions. Some of these

ICPP '19, August 5-8, 2019, Kyoto, Japan
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9999-9/18/06.
https://doi.org/10.1145/3337821.3337867

solutions have also been applied to existing out-of-order processors and GPUs [16, 23]. Typically, caches can provide opportunities for coalescing memory operations in case of reuse, through mechanisms such as the Miss Status Holding Register (MSHR). As discussed above, caches may not necessarily provide benefits for data-intensive, irregular workloads. Additionally, current coalescing solutions are typically tuned for JEDEC DDR compliant protocols, utilizing transactions with a size consistent with cache line sizes (e.g., 64B for general purpose processors and 128B for GPUs).

In the case of 3D-stacked memory, such as HMC, which favors large request packets to achieve the maximum throughput, conventional DMC methodologies are not optimal [12, 23, 26]. These approaches cannot leverage the large request packets provided by the HMC protocol, sacrificing its flexibility and leading to the creation of many small packets with high control overheads and potential to generate a large number of bank conflicts, significantly hindering the bandwidth utilization and overall efficiency of the memory device. We need memory coalescing solutions that are specific to 3D-stacked memory devices, particularly for custom architectures attempting to accelerate data-intensive applications.

Driven by these motivations, we propose and evaluate a scalable memory coalescing infrastructure for 3D-stacked memory devices, named Memory Access Coalescer (MAC). MAC implements a two-phase coalescing mechanism that builds aggregated requests adaptively depending on the requested data (FLow control unITs - FLITs). MAC is adaptable to the majority of modern architectures. Targeting architectures specifically designed to address requirements and behaviors of data-intensive irregular applications, we implement and evaluate the MAC on a custom architecture based on an extended RISC-V [44] Instruction Set Architecture (ISA). We evaluate the MAC design and study its impact from several aspects including latency, bandwidth utilization and request reductions.

The contribution of this research study is three-fold. **First**, we introduce MAC, a novel and generalized FLIT-based memory coalescer. MAC acts as an interface between a custom multicore processor design and the 3D-stacked memory, increasing the bandwidth utilization and thus the performance of data-intensive applications. **Second**, we present the design and implementation of the raw request aggregator based on a FLIT map and the two-stage pipelined request builder with its associated FLIT table. **Third**, we evaluate MAC with representative irregular applications that exhibit a variety of memory access patterns, providing a detailed analysis of the bandwidth utilization, and performance improvement achieved with the MAC.

The paper proceeds as follows. Section 2 provides motivations and background information for this work. It also presents an overview of directly related previous works. Section 3 describes the overall design of the envisioned custom architecture for data analytics, and discusses the operations of the proposed MAC unit. Section 4 introduces the detailed design of the MAC, including the request aggregator, request builder, and FLIT Table. Section 5 provides the experimental evaluation and discusses the findings. Finally, Section 6 concludes the paper.

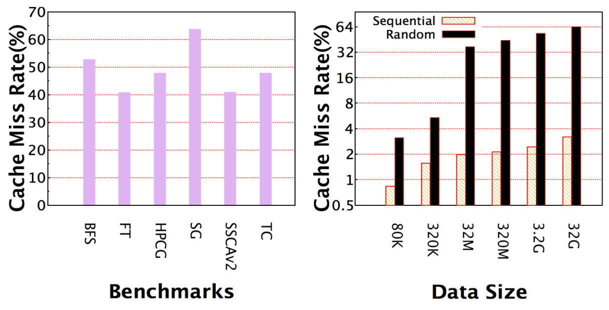


Figure 1: Miss Rate Analysis

2 BACKGROUND

2.1 Irregular Applications

To better understand issues and challenges caused by irregular behaviors of data-intensive applications, we first evaluate several parallel benchmarks with unpredictable data accesses and large datasets. These benchmarks include the High Performance Conjugate Gradient [2] benchmark, the PCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark [8], the NASA Advances Supercomputing (NAS) [9] benchmarks, and the Graph Algorithm Platform (GAP) [11] benchmarks. As illustrated on the left side of Figure 1, the cache miss-rate of these workloads is very high. On average, we see a miss-rate of 49.09%. The Scatter and Gather (SG) and the HPCG benchmarks even reach miss-rates over 50%. Obviously, a miss requires both accessing the main memory and handling the cache miss itself, which induces higher latency than just directly accessing the memory. The high cache miss-rate is a function of both data size and irregular memory access patterns. On the right side of Figure 1, we compare miss-rates for sequential memory accesses (A[i] = B[i]) with random memory accesses (A[i] = B[C[i]], where C[i] is a random positive integer smaller than the size of array B) in the SG benchmark. As the dataset grows from 80KB to 32GB, the cache miss-rate for the sequential memory accesses only increases to 2.36%, while the cache miss-rate for random memory accesses grows from 3.12% to 63.85% (over 20 times). The high miss-rates highlight an extremely low reuse (i.e., locality) of the data loaded in the cache: accesses are all fine-grained, and entire cache lines are replaced to only load and use a single memory word. It is easy to infer significantly worse miss-rates for the irregular workload if the dataset grows to terabytes, which is typically the case for data-intensive irregular applications. As a consequence, it appears clear that, rather than focusing on latency reduction techniques, such as caches, architectures optimized for these applications need to focus on latency tolerance. At the same time, they need to provide enough concurrency to actually allow latency tolerance, and produce enough requests to saturate the available memory bandwidth. However, it also appears clear that if memory transactions are larger than the typical single-word of data requested, much of the available data bandwidth may actually be wasted.

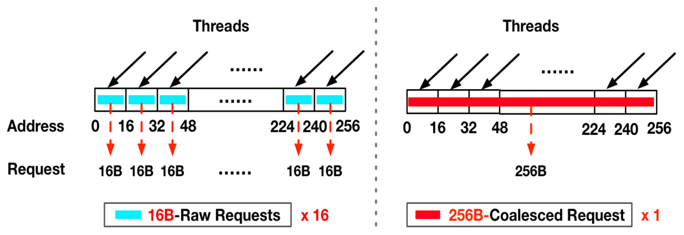


Figure 2: Example of Coalesced and Raw Requests

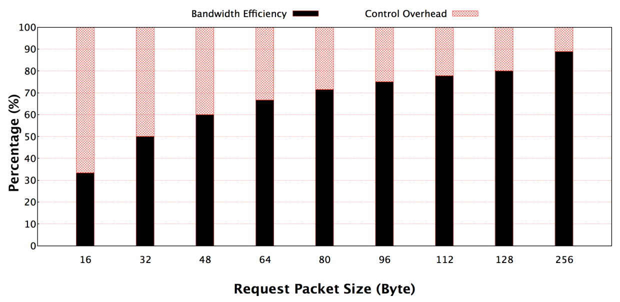


Figure 3: Bandwidth Efficiency and Overhead

2.2 3D-Stacked Memory and Memory Coalescing

In general, Double Data Rate (DDR) RAM devices provide an increase in bandwidth by increasing the n-bit prefetch (where n is 2 for DDR, 4 for DDR2, and 8 for DDR3 and DDR4) of burst accesses. As a results, the minimum access granularity for DDR3 and DDR4, for a typical 64-bit wide data bus, is fixed to 64 Bytes [45].

At the opposite, HMC devices employ a packetized memory protocol that provides flexible request packets varying from 16 Bytes to 256 Bytes [3]. The basic data unit in the HMC protocol is a FLIT (FLow Control unIT) of 16 Bytes. However, the HMC achieves its maximum bandwidth when transmitting large requests (i.e., 128 Bytes and 256 Bytes), and starts to significantly lose efficiency with the smaller ones (i.e., 16 and 32 Bytes) [18, 21, 38]. Hence, approaches such as Dynamic Memory Coalescing (DMC) to aggregate requests are desirable to maximize efficiency.

2.2.1 Bank Conflict Reductions. The conventional row-buffer hit harvesting memory controller is widely used to avoid bank conflicts by prioritizing the row-hit accesses in DDR devices with the open-page policy [37]. Unfortunately, the closed-page memory architecture of HMC does not make possible to aggregate requests at the memory controller level [3]. 3D-stacked memory employs short row lengths (256B and 1KB in HMC and HBM, respectively) to save power. This, in fact, significantly reduces the *overfetch* problem, where many unused bits are brought into the sense amplifiers. However, compared with the 8KB~16KB rows in DDR3, shorter rows reduce the row buffer hit rate, making the open page mode impractical [38, 40]. Furthermore, since each cube is organized in many banks (512 banks in an 8GB HMC), always leaving the DRAM's rows open would lead to high power consumption. Therefore, the DRAM operation in HMC follows a closed-page policy. On completion of a memory reference, the sense amplifiers are precharged and the DRAM row is subsequently closed [20, 24, 33]. With this approach, nearly every memory access is a row buffer miss and cannot be aggregated. Thus, a different processor-side memory coalescing model is desired to reduce bank conflicts and repetitive data transactions in the 3D-stacked memory. For example, suppose that there are 16 threads that simultaneously access data in the same 256B HMC row, as shown in Figure 2. Each thread independently loads a single FLIT (16B) within the row: 0~15B, 16B~31B, etc. If sixteen consecutive 16B load requests are issued to the same 256B HMC row, then the memory controller will send these 16 independent requests to the specific bank, opening and closing the row in the same bank 16 times, as shown on the right of Figure 2. In this case, the bank conflicts occur repetitively and the pipelined memory accesses are executed sequentially, reducing memory-level parallelism and bandwidth utilization [3]. At the opposite, if the raw requests are coalesced, as shown on the right of Figure 2, the single coalesced 256B request will need to access the bank and row once, eliminating 15 bank conflicts. Therefore, DMC may also improve bandwidth and access latency of the 3D-stacked memory by reducing bank conflicts.

2.2.2 Bandwidth Utilization. Besides bank conflicts, we also investigate the bandwidth utilization of the 3D-stacked memory. Each successful data access in HMC involves a pair of request and response packets. These carry not only the actual data, but also the associated control information, such as cube ID, address, tag, cyclic redundancy check, error codes, etc. The control information is stored in headers and tails and attached to both request and response packets. This control information requires 16B per packet (1 FLIT) and 32B per memory access, regardless of the actual data payload [3, 43]. In other words, a 16B and a 256B HMC load request involves the same 32B control overhead. Thus, it is clear that the control overhead can significantly influence the actual bandwidth utilization. Therefore, to express the bandwidth utilization, we define the **bandwidth efficiency** metric as:

$$Bandwidth\ efficiency = \frac{Request\ Size}{Request\ Size + Overhead} \quad (1)$$

Accordingly, we can derive fraction of the overhead as $1-bandwidth\ efficiency$. To quantify the impacts of the request size on the bandwidth utilization, we also compute bandwidth efficiency and overhead for each different possible request size. As depicted in Figure 3, bandwidth efficiency increases as request size increases from 16B to 256B. Conversely, the overhead reduces from 66.66% to 11.11% as the request size increases. Bandwidth efficiency for 256B HMC requests is 88.89%, providing an improvement of a factor of 2.67 when compared with 16B requests. In the example of Figure 2, the dispatching of 16 contiguous requests of 16B requires 768B in total, of which 512B just are control overhead. The single 256B request instead requires 288B, of which only 32B are control overhead. Thus, bank conflicts become extremely detrimental also in terms of bandwidth utilization, further motivating the need to coalesce small requests into larger ones.

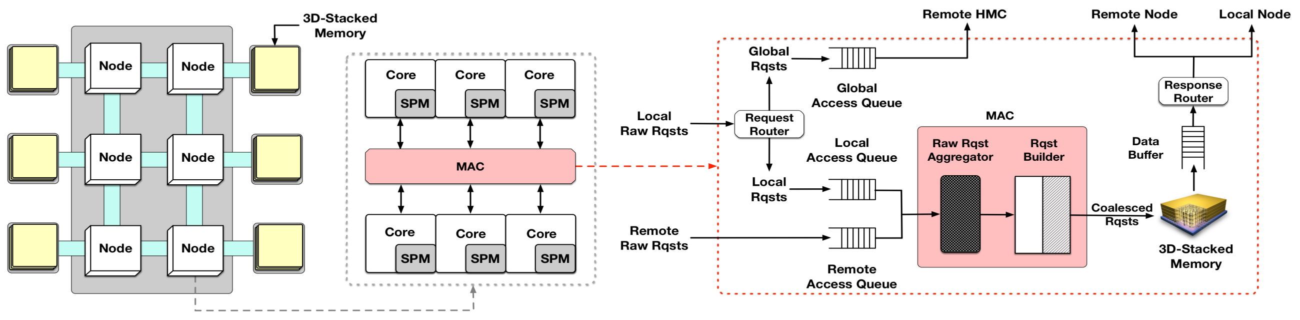


Figure 4: Architecture of Memory Access Coalescer Design

2.3 Dynamic Memory Coalescing

This section briefly summarizes related work in the area of DMC and discusses the limitations of existing approaches.

2.3.1 Related work. Memory coalescing refers to methods that buffer, reorder, and combine a number of memory operations (typically accessing logically or physically close addresses) in a reduced number of larger memory transactions [23]. The majority of modern architectures implements buffer-based coalescing approaches. GPUs typically exploit the L1 data cache to coalesce global memory accesses [4]. GPUs host in their memory hierarchy a *coalescer* unit, which combines accesses to the same 128 Bytes L1 cache line from different threads in a warp. If the L1 experiences a cache miss for the full 128 Bytes line, the request is then partitioned in multiple 32 Bytes transactions that access the respective 32 Bytes L2 cache segments [4]. Some works [26] have proposed extensions to coalesce accesses across threads of different warps. Both CPUs and GPUs can exploit miss status holding registers (MSHRs) to merge outstanding requests of the same cache line, thus reducing the number of accesses to main memory [6]. Original objective of the MSHR is to provide non-blocking cache operations (lock-up free), enabling CPUs to continue executing instruction when handling one ("hit under miss") or more (miss under miss) cache misses [28].

Once a cache miss occurs, the miss handling architecture (MHA) immediately allocates an MSHR entry to hold the request. Any subsequent request to the same block that appears in the stream is then merged with this entry [36], until the original request is not handled. Once the original miss is handled, the corresponding MSHR entry is subsequently freed [14]. Both private and shared caches exploit MSHRs to manage multiple misses without blocking.

2.3.2 Limitations. All the previously described coalescing approaches are either based on the miss handling architecture for memory systems composed of multi-level data caches, and/or specifically target conventional DDR memory interfaces and protocols.

Once a miss occurs in the last level cache (LLC), the MHA allocates a new MSHR entry and immediately dispatches the request to the memory interface. Subsequent requests are then merged during the time the original miss is handled. The request size always is a cache line, no matter how many incoming requests are merged into the same MSHR entry. Cache line sizes in commercial processors typically are 64B, also to be consistent with the burst size of the JEDEC's DDR4 standard. Therefore, coalescing through MHA only allow a fixed size of 64B. This makes difficult to leverage the larger requests packet afforded by the 3D-stacked memory. As discussed in Sections 2.2.1 and 2.2.2, HMC requests of 64B are not ideal, due to the low bandwidth efficiency and increased number of bank conflicts with respect to 256B requests.

However, just enlarging the cache line size to 256B may not be a good choice, especially for irregular applications. In fact, while larger packets maximize the throughput of 3D-stacked memory, they may both waste actual data bandwidth and introduce higher latency. For instance, if every single miss in the a 256B cache line forces a 256B request to the HMC, then each memory request will be 18 FLITs (header + data + tail), even for the single word accesses (64 bits) in typical irregular applications. Lacking any spatial and temporal locality, the rest of the data read (or written) to the memory would just waste usable bandwidth, up to 94.44%.

In conclusion, we argue that a scalable dynamic infrastructure, able to adaptively adjust size of aggregated request depending on the actual data, and specifically targeted to the 3D-stacked memory, is needed to maximize its efficiency with data-intensive, irregular applications.

3 ARCHITECTURE

Figure 4 shows the design of a scalable multi-node, non-uniform memory architecture (NUMA) system targeted at accelerating data-intensive, irregular applications. Each node is an independent processor, directly connected to a 3D-stacked memory device, to provide near memory processing. Nodes can be different tiles of an on-chip design, different dies interconnected together with appropriate interfaces (e.g. through the Intel's Embedded Multi-die Interconnect Bridge - EMIB), or different chips with external interconnect (board or network). Remote 3D-stacked memory devices can be accessed going through the directly attached node. It is not within the scope of this paper to discuss node-to-node communication mechanisms, but we provide the architecture overview to show the generality of our coalescer design. Each node hosts multiple simple in-order cores and integrates a unified MAC (Memory Access Coalescer) through the 3D-stacked memory interface, as shown in the dashed square in the middle of Figure 4. Given the analysis provided in Section2.1 regarding the high cache miss-rates with irregular workloads, nodes do not integrate data caches. Instead, each core presents a directly addressable scratchpad memory (SPM). Unlike caches, SPMs do not have overhead from TLB lookups or tag comparisons, and do not require logic to manage coherency as they are explicitly programmed. This provides reduction in terms of area and power [10, 27, 31], and more opportunities to optimize for the specific behaviors of applications, although it complicates the programming model. As workloads do not benefit from latency reduction, we exploit spatial parallelism to tolerate memory access latencies. Cores will generate memory references and stall until the memory operation completes. Given the high data parallelism of many data-intensive applications, we expect that some of the cores will progress while others are stalled waiting for a memory operation to complete. Note that a natural extension for this architecture is to exploit the scratchpad memory to enable temporal multithreading with quick context switching, in case the architecture is not able to generate enough concurrency to saturate the memory subsystem with just spatial multithreading. The dashed rectangle on the right of Figure 4 details the MAC design.

3.1 Request Router

Given the NUMA architecture of the system, we can classify memory accesses into two categories depending on the source of the request for a given 3D-stacked memory device: local accesses, if memory references are generated from the local node, and remote accesses, if memory references are coming in from a remote node. A request router first analyzes the raw requests generated by the local node. If these requests need to access the local 3D-stacked memory device, the router forwards it to a *Local Access Queue*. If they need to access a remote 3D-stacked memory device, they are forwarded to the respective remote node through a *Global Access Queue*. Another queue, the *Remote Access Queue* collects raw memory requests coming from remote nodes. These queues are FIFO queues appropriately sized that decouple the cores from the actual memory subsystem.

3.2 MAC

The memory access coalescer consists of two components:

1. Raw Request Aggregator. The Raw Request Aggregator is the first stage of the coalescer. It is responsible for grouping small raw requests depending on request types and physical addresses. Raw requests that fall into the same row are merged together in the aggregator and subsequently dispatched to the Request Builder to assemble requests of 3D-stacked memory. Section 4.1 presents the details of the Raw Request Aggregator.

2. Request Builder. The Request Builder prepares the actual request transactions. Given the flexibility of the HMC protocol in terms of packet sizes, it adaptively adjusts the size of the generated HMC transactions, depending on the amount of mergeable memory operations and latency that the architecture affords tolerating. The request builder follows a two-stage pipelined design. The first stage identifies the specific row accessed by FLITs. The second stage determines the packet size of the aggregated request and builds the request, employing the FLIT table described in Section 4.2. The assembled request packets are then dispatched to the 3D-stacked memory.

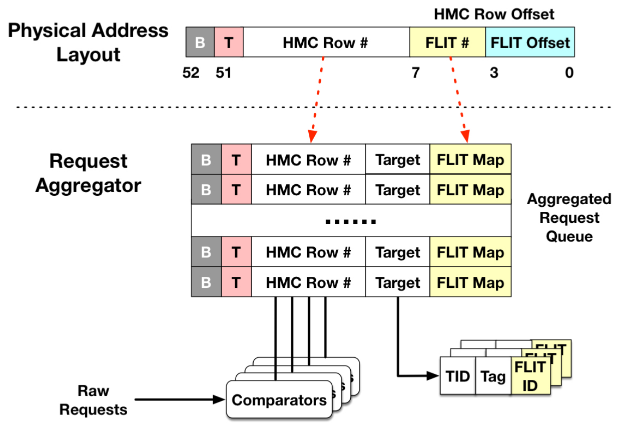


Figure 5: Physical Address Layout and Request Aggregator

3.3 Response Router

Once the local node receives a response from the local 3D-stacked memory device, it stores data into a buffer. The response router analyzes the responses in the buffer and directs the actual data to their destinations depending on the target of each request (e.g. TID, transaction ID, etc.). If the original request is remote, then the response data is routed back to the corresponding remote node through the interconnect. Otherwise, data is directly forwarded to the cores of the local nodes. Afterwards, the target information is utilized to match the pending requests in the load/store queues of each node. Eventually, the cores receive the data and requests are satisfied.

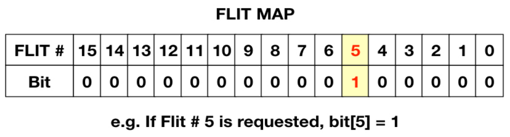
4 MAC DESIGN

As shown in Figure 4, the MAC consists of two main components: the ***Raw Request Aggregator*** and the pipelined ***Request Builder***. The following two subsections describe each component in detail.

4.1 Raw Request Aggregator

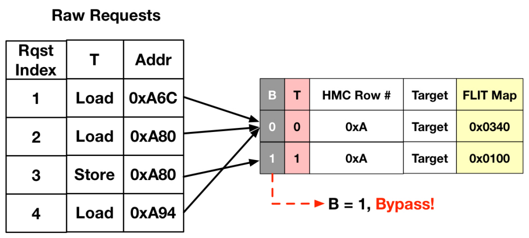
As shown at the top of Figure 5, we can partition the physical address of each request into two segments: *row number* and *row offset*. To allow reaching the peak bandwidth when possible, the MAC design considers an HMC device configured with the largest row size, 256B. Thus, the lower 8 bits of the physical address serve as the row offset to access the data inside of each DRAM row. Similarly, the address bits for DRAM, bank and vault are combined in the row number that specifies the index of each HMC row. Since transactions in the HMC protocol have a minimum granularity of one FLIT (16B), we can ignore the FLIT offset placed in bits 0~3 and use bits 4~7 to record the requested FLIT number.

The Raw Request Aggregator implements a First In First Out (FIFO) queue, named *Aggregated Request Queue* (ARQ), that holds pending requests to the 3D-stacked memory, as shown in Figure 5. Each ARQ entry is associated with a comparator. Given the interleaved vaults in HMC, we consider the DRAM row as the unit and coalesce the requests falling into the same row to enhance the memory-level parallelism (MLP) and reduce bank conflicts. Once a raw request enters the request aggregator, the HMC row numbers


FLIT MAP
e.g. If Flit # 5 is requested, bit[5] = 1

FLIT #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Bit	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0

Figure 6: The Flit Map Layout



Rqst Index	T	Addr
1	Load	0xA6C
2	Load	0xA80
3	Store	0xA80
4	Load	0xA94

B	T	HMC Row #	Target	FLIT Map
0	0	0xA	Target	0x0340
1	1	0xA	Target	0x0100

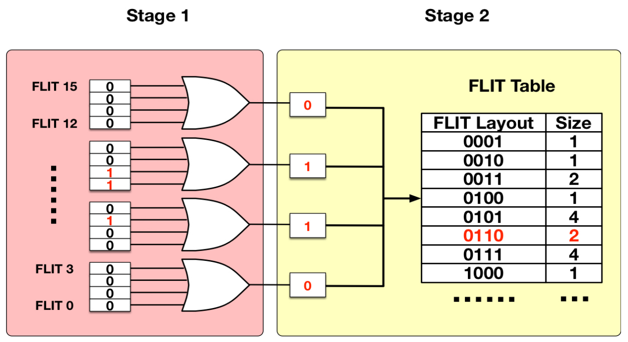
Figure 7: Example of Request Aggregations

of the incoming request and all currently pending requests in the ARQ are simultaneously compared. If there is a hit, then the new request is merged into the respective entry in the ARQ. Otherwise, a new entry is assigned to hold the incoming request, registering its row number.

In addition to load and store requests, the request aggregator also handles memory fences. Once a memory fence operation enters the ARQ, comparators are immediately disabled, and all the following requests can only be stored into subsequent free entries. This means that the request aggregation is disabled until the memory fence operation is popped out, implying that all previous memory requests have been executed.

Every two clock cycles, a request is popped out from the ARQ and delivered to the Request Builder to construct the HMC request. We also introduce a latency hiding mechanism to support I/O-bound workloads or allow booting new programs. The ARQ includes a counter that records currently available entries. If the counter reaches a value N larger than half of the ARQ size, then the N following raw requests from either the local or the remote access queue will bypass the hardware comparators and directly use all available ARQ entries. In this way, we can ensure a sufficient amount of requests in the ARQ to perform aggregation.

4.1.1 FLIT Map. The Raw Request Aggregator also hosts a bitmap-based structure, named *FLIT map*, to record which FLITs have been requested in each DRAM row. As shown in Figure 6, the FLIT map contains 16 bits (one bit per FLIT) and all bits are initialized to 0. If a memory operation accesses one FLIT of the current HMC row, the Raw Request Aggregator will set the respective bit to 1. For example, in Figure 6, we see that FLIT number 5 has been requested, thus $bit[5]$ is set to 1. The target information of the coalesced requests are then stored in the *target* segment. These include thread ID, transaction tag, and ID of the requested FLIT. In the proposed design, the TID and tag uses 2B each, allowing to handle up to 64K threads and 64K transactions per thread. Since each HMC row only contains 16 FLITs, 4 bits are sufficient to hold the FLIT ID of each request. As such, each target requires a total of 4.5B. Section 4.4 presents a detailed evaluation of the area overhead.



FLIT Layout	Size
0001	1
0010	1
0011	2
0100	1
0101	4
0110	2
0111	4
1000	1
......	...

Figure 8: Two-Stage Pipelined Request Builder

4.1.2 Address Extensions. To facilitate request aggregation, the Raw Request Aggregator also employs two additional bits that extend the addresses of the requests, identified with letters B and T in Figure 5. The B bit signals bypassing. If there is only one request falling into a DRAM row, then the B bit of that entry is set. This indicates that the Raw Request Aggregator cannot coalesce any other request in the same row. Otherwise, the B bit is unset. The Raw Request Aggregator checks the bypassing bit before popping out requests from the ARQ. When B is set, the Raw Request Aggregator directly forwards the request to the 3D-stacked memory, bypassing other stages of the MAC. Otherwise, the aggregated requests are routed to the Request Builder.

The T bit indicates request types, simplifying the comparison process. Obviously, requests of different types (load and store) cannot be coalesced. However, comparing request addresses and request types separately would have higher latency than a single comparison. Thus, we use the T bit to distinguish loads (0) from stores (1). We consider that current 64-bit architectures use up to 52 bits (0∼51) to represent physical addresses, and we use bit 52 to identify the type of memory request. Thus, the added T bit has highest bit of incoming request addresses (row number) in the ARQ, allowing to distinguish stores from loads quickly with a single comparison. In addition, the atomic operations are directly routed to the 3D-stacked memory to ensure the atomicity, which will not be coalesced.

Figure 7 shows an example with four raw requests. Requests 1, 2 and 4 are load requests hitting the same row $0xA$. The Raw Request Aggregator merges these three requests into the same ARQ entry, and sets the bits for FLIT 6, 8 and 9 to 1 in the FLIT map. The operation in request 3 is, instead, a store, thus the corresponding T bit is set to 1. Therefore, it is placed in a separate entry of the ARQ, even if it accesses the same row as requests 1, 2, and 4. Since the Raw Request Aggregator cannot merge any additional request to request 3, the B bit is set, triggering bypassing.

4.2 Pipelined Request Builder

As discussed previously, while the HMC reaches its maximum bandwidth with the largest packet size (256B), it would waste significant bandwidth to issue transactions of such size always for memory operations that actually require only a small portion of the HMC row. Thus, if it is not possible to coalesce as many operations, an efficient coalescing unit for the HMC should trade off some of the control overhead (i.e. use smaller packets) to increase actual utilization of the data bandwidth. The two-stage pipelined Request Builder of the MAC, which dynamically constructs request transactions on the base of the actual memory requests, aims at mitigating this issue. We configure the Request Builder to generate requests from 64B to 256B, as a trade-off between control overhead and data utilization.

Figure 8 shows a schematic of the Request Builder. The first stage takes the FLIT map of a HMC row request from ARQ as input, and evenly partitions the 16-bit FLIT map into 4 groups. Each group represents a (consecutive) chunk of data (64B). The first stage of the Request Builder then performs a logic OR operation among these 4 bits of each group. This only requires a single cycle. The results (4 bits) are then forwarded to the second stage.

4.2.1 FLIT Table. The second stage of the Request Builder generates coalesced requests with proper sizes via the FLIT table. In our implementation, the FLIT table is a simple look-up table [42], composed of two columns. The first column simply lists all possible combinations of 4 bits, which corresponds to these 4 bits computed in the first stage of the Request Builder. Each bit represents 64B of data, i.e. the minimum granularity for transactions emitted by the MAC. Obviously, if one bit is set, the respective 64B data chunk contains one or more active FLITs request. If there are no active FLIT requests in the data chunk, then the bit remains unset. The second column associates the corresponding size of the coalesced request transaction depending on the permutation of these 4 bits (1, 2 and 4, respectively identifying sizes of 64, 128, and 256B). The use of this small look-up table constraints the latency of the second stage of the Request Builder to only two cycles: one for the table look-up, and one for building the final request. The space overhead is also minimal (12B for the 16-entry look-up table). Continuing with the example in Figure 7, the FLIT map of the coalesced raw requests {1, 2 and 4} produces the 4-bit sequence 0110. The Request Builder then builds and issues a 128B aggregated transaction, according to the FLIT Table of Figure 8,

4.3 Applicability

The MAC infrastructure provides the portability to different HMC generations. As the transition from HMC 1.0 to HMC 2.1 brought an increase of the maximum HMC request size from 128B to 256B, we may expect maximum request size to be increased even more in the future. The proposed MAC design is general enough to support larger requests by simply enlarging the FLIT map and the FLIT table. While this work focuses on HMC (considered as a component-off-the-shelf for the design of the architecture for irregular applications presented in Section 3), the MAC design is also applicable to High Bandwidth Memory (HBM). HBM employs traditional DDR protocols that transfer data via a burst mode. The burst length (BL) determines the granularity of data transactions, whereby *BL* is 2 in DDR, 4 in DDR2/HBM, and 8 in DDR3/DDR4 [45]. As such, with a typical 64-bit wide data bus, the memory access granularity in HBM is fixed to 32B [1, 34, 41], which is identical to a 2-FLIT transaction in HMC. Given the 1KB page size (i.e. row size) in the HBM [1], the requests coalesced by MAC (64B∼1KB) may require 2 ∼ 32 bursts in HBM (4 ∼ 64 FLITs). Therefore, MAC can be applied to HBM by changing the protocol, but without modifying any of the associated coalescing design and logic.

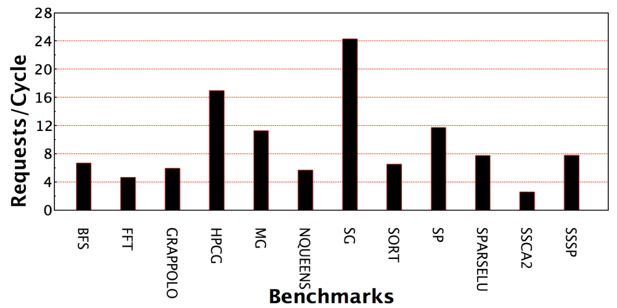


Figure 9: Raw Requests per Cycle

4.4 Latency Analysis

As discussed in Section 4.2.1, since the latency of Request Builder pipeline is two cycles, the issuing rate of the MAC is fixed to 0.5 requests per cycle. Since the ARQ accepts one request per cycle, once there is more than one raw request available to enter the the ARQ, we can issue memory transactions at a constant rate from the Request Builder.

$$RPC = IPC \times RPI \times \# \ of \ Cores \times Mem\_Access\_Rate \tag{2}$$

We briefly discuss the memory request rate of each node. We can derive the number of requests per cycle (RPC) from Equation 2, where IPC and RPI represent the Instructions Per Cycle and the Requests Per Instruction, respectively. The mem_access_rate is the average number of memory operations to the MAC (i.e. the memory operations for which data are not in the private SPMs of our node architecture). Following this metric, we measure the respective IPC, RPI, and Mem_Access_Rate assuming 8 cores and a CPU frequency of 3.3 GHz (we detail the experimental setup in Section 5.2). We then compute the corresponding RPC for each benchmark. As shown in Figure 9, all benchmarks are able to provide MAC with more than 2 raw requests per cycle. On average, there are up to 9.32 new requests per cycle ready to enter the ARQ, thus providing enough concurrency to saturate memory links and perform aggregation.

5 EXPERIMENTAL EVALUATION

5.1 Simulation environment

We implement and simulate the MAC using an environment based on RISC-V cores with the RV64IMAFDC ISA. Our simulation environment consists of various components. We first implemented a memory tracer that captures the memory operations generated by simulating a multiprocessor system with the RISC-V ISA simulator (Spike). Each memory instructions is associated with originating thread and core (target information). We implemented the SPMs in the RISC-V Spike simulator and extended the RISC-V ISA as well as the associated RISC-V cross-compiler (riscv64-unknown-linux-gnu-gcc) to enable functionality (prefetching, write-back, etc.) of the software-managed SPMs. We then designed an analyzer that inspects the memory instruction stream and retrieves HMC row number and FLIT ID. Finally, we implemented the actual timed MAC simulator, which models its components and its behavior. It takes as input the memory instruction stream and the processed

Table 1: Simulation Environment Configurations

Parameters	Value
ISA	RV64IMAFDC
Core #	8
CPU Frequency	3.3 GHz
SPM	1MB per core
Avg. SPM Access Latency	1 ns
HMC	4 Links, 8GB, 256B-block
Avg. HMC Access Latency	93 ns
ARQ	32 entries, 64B per entry

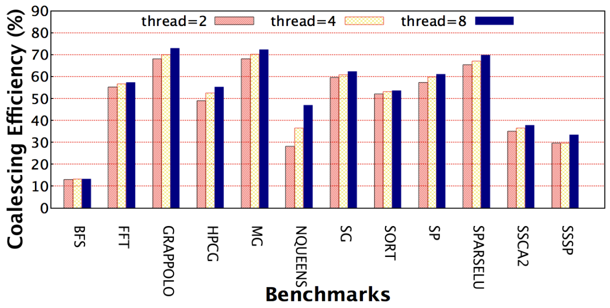


Figure 10: Coalescing Efficiency

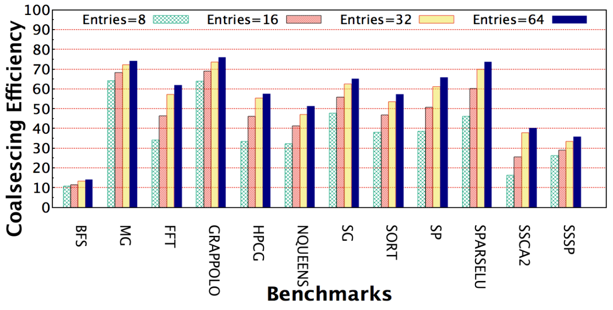


Figure 11: The Impacts of ARQ Entries

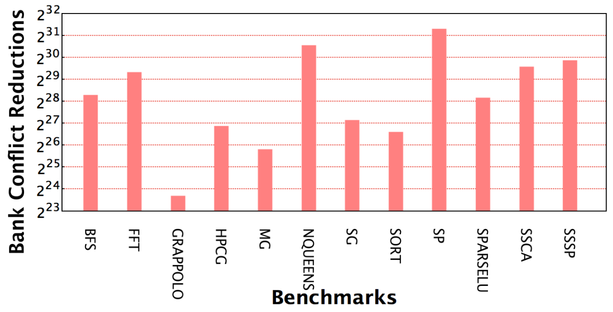


Figure 12: Bank Conflicts Reductions

HMC information, and simulates the behavior of the MAC. Our simulation model inputs the generated HMC requests to HMCSim-3.0 [30]. HMCSim provides back the responses to our simulator, which proceeds by retrieving target information and feeding the data back to the originating memory instruction.

5.2 Benchmarks and parameters

We select 12 parallel benchmarks to evaluate the performance of proposed MAC infrastructure. These mostly pertain to the area of data-intensive applications and exhibit a variety of memory access patterns. The benchmarks include: Scatter/Gather (SG), HPCG [2], SSCA2 [8], Grappolo (a parallel implementation of graph clustering using the Louvain method) [32], the Graph Algorithm Platform (GAP) [11], the Barcelona OpenMP Tasks Suite (BOTS) [15], and the NAS Parallel Benchmarks (NAS-PB) [9]. We compiled the benchmarks with the RISC-V GCC 7.1 cross-compiler and executed them on the standard RISC-V Linux system image running on top of our modified RISC-V *Spike* simulator.

Table 1 shows the overall configuration of our simulation infrastructure. We assume 8 RISC-V cores with 1MB SPM/core and a 8GB HMC device configured with 256B HMC rows. Additionally, we configure the ARQ in MAC with 32 entries and allocate for each entry 64 Bytes of space.

5.3 Results and Analysis

In this section we investigate the effects on memory transactions in the 3D-stacked memory provided by MAC.

5.3.1 Request Reductions. To quantify the benefits of MAC, we define *coalescing efficiency* as the ratio between the number of *coalesced* requests (with MAC) and the original number of raw requests (without MAC), as shown in Equation 3. As illustrated in the Figure 10, MAC achieves high coalescing efficiency across all the benchmarks with different concurrency. Following the growth of thread numbers, we observe the increasing trend of the coalescing efficiency from 48.37%, 50.51% and 52.86%, when executing with 2, 4 and 8 threads, respectively. Particularly, with 8 threads, MAC coalesces over 60% requests for MG, GRAPPOLO, SG, SP and SPARSELU. On average, the coalescing efficiency is 52.86%, i.e., MAC coalesces over half of the raw requests.

$$Coalescing\ efficiency = \frac{Requests\ with\ MAC}{Requests\ without\ MAC} \tag{3}$$

We also explore effects of different ARQ sizes (in terms of number of entries). Figure 11 shows that coalescing efficiency increases from 37.58% to 56.04% as the number of ARQ entries increases. We also see, however, a diminishing return: as the number of ARQ entries doubles, the coalescing efficiency does not grow at the same rate. Coalescing efficiency only improves 22.11%, 15.72% and 5.53% as the number of ARQ entries respectively increases to 16, 32 and 64. As such, the 32 entries configured for the simulated environment appears a reasonable tradeoff between efficiency and MAC size.

Since bank conflicts are one of the key factors that hinders memory performance as discussed previously, we need to evaluate the effects of MAC on bank conflicts. Figure 12 shows the reduction in bank conflicts provided by the proposed coalescing unit. Overall, the bank conflicts are reduced by approximately 644 million on average and 7.73 billion in total. It is worthy of note that MAC removes

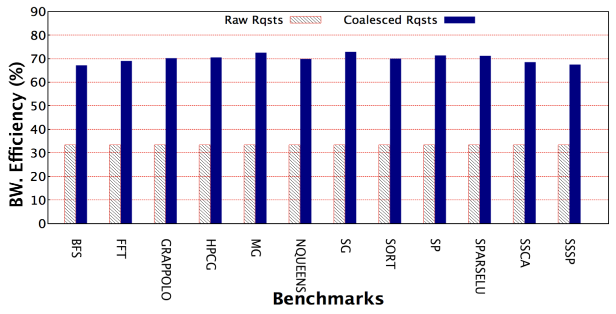


Figure 13: Bandwidth Efficiency

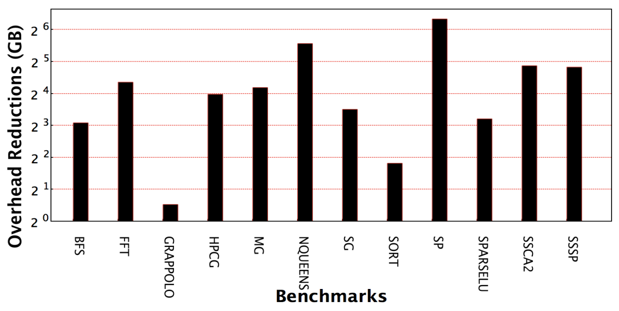


Figure 14: Bandwidth Saving

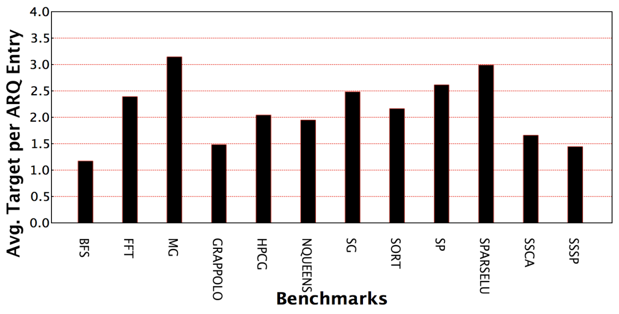


Figure 15: Avg. Targets per ARQ Entry

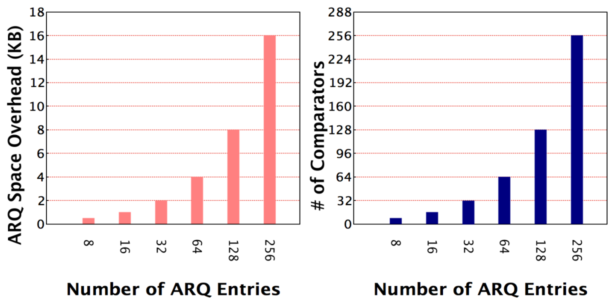


Figure 16: Space Overhead

over 157.24 and 264.64 million of bank conflicts with NQUEENS and SP, respectively.

5.3.2 Bandwidth Utilization. Besides coalescing efficiency, we also evaluate improvements to bandwidth utilization brought by MAC. We first measure the bandwidth efficiency (defined in Equation 1) of the coalesced request transactions, and then compare it with the transmissions of raw requests. As shown in Figure 13, the average bandwidth efficiency of coalesced accesses is 70.35%, while the 16B raw requests only obtain 33.33%. Thus, we see a more than 2 times improvement in bandwidth efficiency. In other words, MAC reduces the control overhead from 66.67% to 29.65%. We also measure total overhead reduction due to request aggregation. Figure 14 shows that MAC saves an average of 22.76 GB of bandwidth for control, thus improving the bandwidth utilization for data.

5.3.3 Overhead Analysis. In this section, we provide a quantitative evaluation of the area occupied by MAC. Since the 64-bit physical address (including the extended two bits) and the 16-bit FLIT map only occupy 10B of the 64B ARQ entry, there are 54B left to buffer the target information of merged requests. Since each request target requires 4.5B, as discussed in Section 4.1.1, the 64B entry is capable of merging up to 12 distinct requests within the same row. To accurately measure the actual space overhead of the request targets, we record the average targets per entry for each benchmark. As shown in Figure 15, the average targets per ARQ entry are much smaller than the target limits (12). On average, approximately 2.13 targets are merged in each ARQ entry, i.e., each entry coalesces 2.13 requests. Since the largest targets per entry among all the tests are 3.14, 54B are enough to store the targets in each ARQ entry.

Regarding the area occupation of the Request Aggregator, as shown in Figure 16, the ARQ size expands from 512 Bytes to 16 KB as the number of ARQ entries increases from 8 to 256. Since each entry is associated with a comparator, the respective space complexity also is $O(n)$. As discussed in Section 4.2, the space overhead of the pipelined request builder is fixed to 14B, needed to implement the FLIT map and FLIT Table. Thus, the total space overhead of the MAC (with the 32-entry ARQ) is a memory of 2062 Bytes, 32 comparators and 4 OR gates. This is comparable to a fully associative cache composed of 32 lines of 64B.

5.3.4 Performance. We finally provide a performance analysis by measuring the difference in execution latency of HMC memory transactions for all applications, as measured by HMCSIM with and without MAC. While this analysis provides understandings of performance improvements in the memory subsystem for memory-bound applications (i.e. applications for which the instructions exhibit an overwhelming amount of memory operations), this approach also well approximates the effective overall performance gain. Figure 17 shows that, for our set of applications, MAC provides an average gain of 60.73% (i.e. the overall latency to access memory is reduced). More specifically, MAC improves the performance of the memory system by over 70% for MG, GRAPPOLO, SG and SPARSELU.

6 CONCLUSIONS

This paper presents MAC (Memory Access Coalescer), a novel memory coalescer for 3D-stacked memory devices. We discussed the

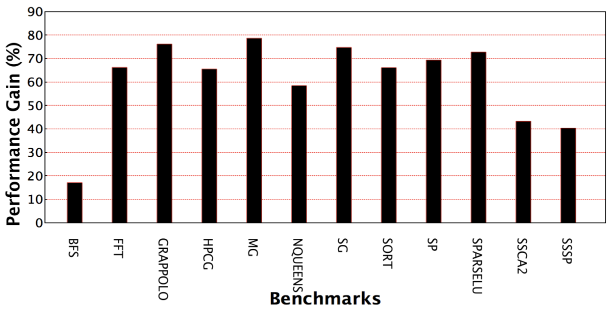


Figure 17: Memory System Speedup

rationale and motivations behind the coalescer, describing a cacheless strawmen architecture focused at accelerating data-intensive irregular applications. We detailed the design of the various components of MAC, including the raw request aggregator, the FLIT maps, the pipelined request builder, and the FLIT table. Using a custom simulation infrastructure based on RISC-V, we have shown that MAC is able to significantly reduce the number of memory transactions and bank conflicts in the 3D-stacked memory over a variety of data-intensive applications. MAC adopts an adaptive memory coalescing approach that exploits the variable size of the request transactions, dynamically selecting sizes that increase actual bandwidth utilization. We study the impacts of MAC from a various points of views, including coalescing efficiency (reduction of the number of memory transactions), bandwidth utilization, overhead and performance improvement, for a large set of irregular applications. Among these benchmarks, MAC coalesces 52.86% of the original memory accesses and improves the performance of the memory subsystem by 60.73%.

REFERENCES

[1] JEDEC Standard High Bandwidth Memory(HBM) DRAM Specification. Technical report, 2013.

[2] Toward a New Metric for Ranking High Performance Computing Systems. Technical report, Sandia National Laboratories, 2013.

[3] HMC Specification 2.1. Technical report, December 2015.

[4] CUDA Toolkit Documentation. Technical report, July 2018.

[5] S. Aga and S. Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *ISCA 2017.*

[6] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler. Selective gpu caches to eliminate cpu-gpu hw cache coherence. In *HPCA 2016.*

[7] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *ISCA 2015.*

[8] D. Bader and K. Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. *HiPC 2005.*

[9] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS parallel benchmark results. SC 1992, Los Alamitos, CA, USA.

[10] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *CODES 2002.*

[11] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[12] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *SC 2011.*

[13] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, 2016.

[14] H. Dai, C. Li, H. Zhou, S. Gupta, C. Kartsaklis, and M. Mantor. A model-driven approach to warp/thread-block level gpu cache bypassing. In *DAC 2016.*

[15] 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 *ICPP 2009.*

[16] N. Fauzia, L.-N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *CGO 2015.*

[17] M. Gao, G. Ayers, and C. Kozyrakis. Practical near-data processing for in-memory analytics frameworks. In *PACT 2015.*

[18] M. Gokhale, S. Lloyd, and C. Macaraeg. Hybrid memory cube performance characterization on data-centric workloads. In *IA3 2015.*

[19] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ICS 2014.*

[20] 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 *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, pages 66–75. IEEE, 2017.

[21] R. Hadidi, B. Asgari, J. Young, B. A. Mudassar, K. Garg, T. Krishna, and H. Kim. Performance Implications of NoCs on 3D-Stacked Memories: Insights from the Hybrid Memory Cube. *arXiv preprint arXiv:1707.05399*, 2017.

[22] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 2016.

[23] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *TPDS*, 2011.

[24] J. Jeddeloh and B. Keeth. Hybrid memory cube new DRAM architecture increases density and performance. In *VLSIT 2012.*

[25] G. Kim, J. Kim, J. H. Ahn, and J. Kim. Memory-centric system interconnect design with hybrid memory cubes. In *PACT 2013.*

[26] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke. Warppool: sharing requests with inter-warp coalescing for throughput processors. In *MICRO 2015.*

[27] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have your scratchpad and cache it too. In *ACM SIGARCH Computer Architecture News*, 2015.

[28] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA 1981.*

[29] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *MICRO 2010.*

[30] J. D. Leidel and Y. Chen. HMC-Sim: A simulation framework for hybrid memory cube devices. *Parallel Processing Letters*, 24(04):1442002, 2014.

[31] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou. Understanding the tradeoffs between software-managed vs. hardware-managed caches in gpus. In *ISPASS 2014.*

[32] P. Mahantesh Halappanavar. Grappolo. Technical report, Pacific Northwest National Laboratory, 2014.

[33] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, and C. Evangelinos. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17–1, 2015.

[34] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. In *MICRO 2017.*

[35] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *ICS 2014.*

[36] P. Prieto, V. Puente, and J. A. Gregorio. CMP off-chip bandwidth scheduling guided by instruction criticality. In *ICS 2013.*

[37] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *ISCA 2000.*

[38] P. Rosenfeld. *Performance exploration of the hybrid memory cube.* PhD thesis, 2014.

[39] L. Schares, B. G. Lee, F. Checconi, R. Budd, A. Rylyakov, N. Dupuis, F. Petrini, C. L. Schow, P. Fuentes, and O. Mattes. A throughput-optimized optical network for data-intensive computing. *IEEE Micro*, 2014.

[40] J. Schmidt, H. Fröning, and U. Brüning. Exploring time and energy for complex accesses to a hybrid memory cube. In *MEMSYS 2016.*

[41] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. MemZip: Exploring unconventional benefits from memory compression. In *HPCA 2014.*

[42] D. Shin, J. Lee, J. Lee, and H.-J. Yoo. 14.2 dnpu: An 8.1 tops/w reconfigurable cnn-rnn processor for general-purpose deep neural networks. In *ISSCC 2017.*

[43] X. Wang, J. D. Leidel, and Y. Chen. Memory coalescing for hybrid memory cube. In *ICPP*. ACM, 2018.

[44] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.7. Technical Report UCB/EECS-2015-49, May 2015.

[45] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *ISCA, 2011.*

[46] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: throughput-oriented programmable processing in memory. In *HPDC 2014.*