15-740 Project Report
In-Memory Data Filtering for Cuckoo Hashing

Pooja Nilangekar, Tushar Agarwal and Vamshi Reddy Konagari
POOJAN, TAGARWA1, VKONAGAR @ andrew.cmu.edu

1 Introduction

The process-memory gap – the fact that DRAM memory performance is significantly insufficient to service processors' requests at the required rate – poses one of the most significant challenges to achieving peak performance in contemporary applications [29, 25, 35]. Recent innovations in DRAM hardware include integration of a layer of logic to create special memory-chip devices from 3-D stacked memory [25, 26, 36] using Through-silicon vias (TSVs) to communicate at a significantly higher bandwidth, lower latency and using lesser energy than the processor memory bus (figure 1).

Therefore, it is now not only beneficial to offload parts of computation in applications to DRAM's side, but PIM integration is also an architectural change to which real applications may need to adapt to soon [1, 2]. Cuckoo Hashing, a variant of hashing with numerous real-world applications including Open vSwitch [30] and Memcache [10], is one such application which can enjoy improved performance – both concerning execution speed and energy requirement – from PIM.

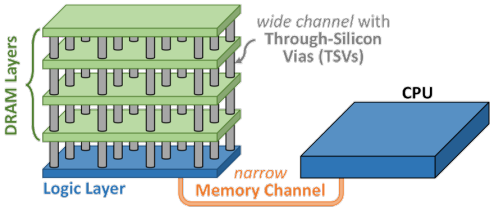

Figure 1: A layer of compute integrated into 3-D stacked memory using Through-silicon vias (TSVs). Image source: [11].

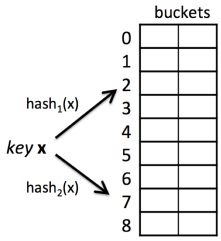

Figure 2: A Cuckoo hash can map a key-value pair to at most two buckets. Image source: Google Images.

Cuckoo Hashing is a memory efficient – in terms of storage space – hashing technique which guarantees $O(1)$ insertion and lookup times amortized across multiple accesses [28]. Variants of Cuckoo hashing can handle a higher load factor than standard hash tables and hence can work with smaller tables than non-Cuckoo hashing for the same number of key-value pairs. A Cuckoo hash uses an array of buckets, each with multiple slots to store a key, value, and hash. A key-value pair can be mapped to at most two buckets (figure 2). To lookup a key, then, we need to fetch both buckets from memory and check hashes in slots in both buckets. If the hash value at a particular slot matches with the query hash, we check if the query key matches with the key stored in the slot. If both match, the key-value pair is returned.

We utilize Processing In Memory (PIM) for efficient retrieval of the key-value pair. The goal is to perform the key-value lookup in memory – more precisely, the logic layer integrated with the DRAM, which we will refer to as a `PIMcore` – instead of fetching the corresponding buckets into the cache. In other words, we offload the computation involved in key matching to the `PIMcore` to improve memory utilization. Once the `PIMcore` receives information about the key to lookup, it will compute the hash value and try to match it with the hash values in the stored bucket slots. If the hashes match, the `PIMCore` will try to match keys and return the matched key-value pair if successful. Even without

coherence support for caches, we can assume, for an immutable hash table, that in-memory matching will return a valid result even if a part of the bucket is present in the CPU cache.

In essence, we plan to filter the key-value pairs to be returned to the core to realize three main objectives:

1. Avoid cache pollution caused by fetching buckets by performing the match in memory and returning only the relevant key-value pair.
2. Decrease the traffic on the processor-memory interconnect.
3. Reduce the power consumption by reducing interconnect use.

2 Related Work

Although there is a trove of recent research on PIM, general techniques and principles related to it date back to more than a decade ago. Most of early research recognized PIM's potential to improve latency and bandwidth to main memory and focused on performance improvements for specialized tasks such as computer graphics, parallel computation [16, 17], bioinformatics [15], irregular applications – operations on sparse matrices [13, 8] or N-body computations [6] – or applications on special-purpose computing devices such as supercomputers [12, 22, 5] and embedded systems [23]. As a part of our analysis will also reflect, a system with PIM has important implications for software and architecture design, including support for specific data layout inside the memory [9, 21]. Newer research continues to consider implications of PIM on performance, software and architecture of computer systems [34, 37, 11] and applications – such as multimedia processing [18] – but also reflects a focus on integration with contemporary multiprocessor systems, such as addressing issues related to coherence [4].

We consider our study to be an exploration of PIM for a specific application (Cuckoo hashing), and we draw inspiration, particularly concerning system architecture and memory layout, from recent research on in-memory bulk data copying [32], bitwise operations [33], pointer chasing [14] and DNA filtering [19]. [32] and [33] are detailed treatments on adapting DRAM's internal instructions to perform useful computation, while [14] also performs address translation in PIM-based DRAM, and all three observe significant latency, bandwidth, and energy improvements over application versions using the usual memory hierarchy and the communication bus between the processor and the DRAM. [19] utilizes these concepts for a higher-level filtering task based on matching a small pattern with contiguous substrings at each index in a long string. [32] and [33] demonstrate that bulk-copy and bitwise `AND`, `OR` and `NOT` operations can be performed by executing a series of row access strobes one after another, without intervening row precharge operations. [14] extends the same general principles to speed-up a relatively higher level application – pointer chasing – by using a memory with an integrated processing element, while also implementing operating system structures such as page tables on the side of the main memory. In our study, we extend inferences about memory latency, bandwidth utilization, and energy consumption in the aforementioned research to PIM-based Cuckoo hashing, while also measuring the benefits to applications concurrently executing with the hashing program.

3 Goals

Our most important goal in this study was to explore the design space and trade-offs in the structure of a PIM architecture and its integration with Cuckoo hashing. We described in section 2 several studies which exploit the high bandwidth provided by PIM-based cores to speed up memory-intensive computations. Our project explores how similar ideas can be applied in the context of Cuckoo hashing.

We use the ZSim simulator [31] to model our design choices in detail. Beginning with our goal to model PIM functionality on ZSim to a high level of detail, we iterated through several designs with varying levels of functionality. We began with a simple design where both the main processor and the PIM-based processor were single core in-order processors, with the only difference between the two processors being differences in main memory access. The main processor accessed the memory via the cache hierarchy. On the other hand, the PIM-based processor accessed main memory directly, without any caches coming into play, and with a lower latency due to the existence of TSVs. This design was eventually improved – with the degree of faithfulness to a real PIM-based computer system being the main criterion – to the architecture described in section 5. This architecture enabled the PIM cores to access the memory with high bandwidth and low latency, while also taking into account nuances for PIM logic in memory. We describe these in more detail in section 5.

As part of our final goal, we benchmarked the performance of our PIM architecture on a simple array filtering program which scans a vast array to find a queried element and used this to check the correctness of our architecture simulation. We then executed YCSB [7], a Cuckoo hash benchmark. Finally, to gauge the possible impact on the degree of contention in the last level cache (LLC), we measured the interaction between programs using the PIM-based processor with other programs executing concurrently on the main processor.

4 A note on Architecture Simulators

Several candidate simulators may be used to model a PIM-based architecture. We briefly describe three below.

IMPICA [14], a pointer chasing accelerator, utilized a `Gem5`-based simulator. `Gem5` is a highly detailed full system simulator with various DRAM modules such as `DRAMSim2` [Insert more examples here]. However, since `Gem5` simulates a complete operating system, it is usually slow to boot up ($\approx$ 20 minutes to boot up a full Linux kernel) and compile, and significantly tricky to debug (adding features may require making modifications to several places in the kernel source).

Userspace simulators, such as ZSim [31] and Ramulator [20], based on trapping of application instructions (usually through Intel's `PIN` [27]) to perform simulation present an attractive alternative. Although these simulators, considering their userspace-based nature, are not as faithful as `Gem5`-based simulators to real architectures, they provide several necessary features, including extensibility for adding PIM support, userspace simulation to enable convenient benchmarking of Cuckoo hash programs, low execution overhead and easy debugging. Ramulator [20] is a dedicated DRAM simulator with fine-grained state machines to module internal DRAM latencies accurately. However, it does not model architectural details including the cache hierarchy, which is critical for our study. On the other hand, although ZSim does not model internal DRAM latencies out-of-the-box, it models the cache hierarchy and is extensible enough to allow the addition of the required degree of detail in modeling latency for PIM and a 3-D stacked DRAM.

We chose to utilize ZSim for this study.

5 Architecture

We choose the specifications of Micron's Hybrid Memory Cube (HMC) [3], a 3-D stacked memory variant with an integrated logic layer, to model the DRAM in our PIM-based architecture `PIM DRAM`. Full specifications for the HMC were released to the public by the HMC Consortium, making the HMC

specification sheets an attractive source of fine-grained latency values for modeling the `PIM DRAM`. The `PIMcore` is modeled as a purely in-order core with the same clock speed as the main processor.

5.1 Overview

Our PIM-based system's architecture can be divided into two major parts: (i) the host side and (ii) the memory side (figure 3).

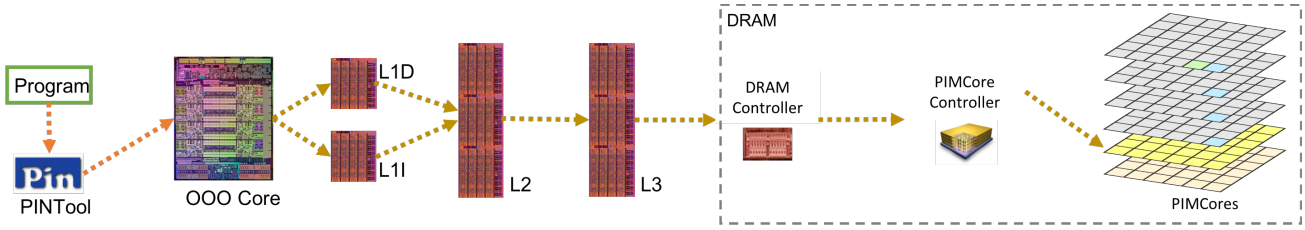

Figure 3: The architecture digram showing PIM DRAM and PIM Cores

The host side comprises a quad-core Intel i7-4770 Haswell processor clocked at 3.4 GHz (Turbo Boost off) with 32 KB L1, 256 KB L2 and 8MB L3 caches (table 1). Although the processor has four cores, we only make use of a single, on which we schedule one or more tasks to execute concurrently.

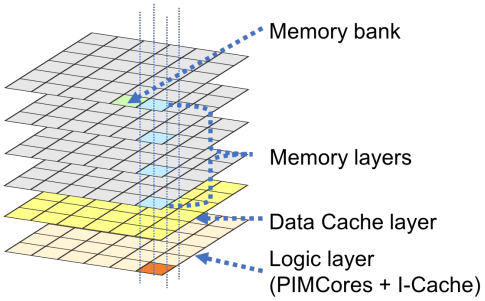

Figure 4: A 3-D stacked memory-based DRAM with integrated logic and cache layers. The blue vertical lines demarcate a vault.

5.2 PIM DRAM

The memory side consists of a 3-D stacked memory-based DRAM with an integrated logic layer based on the 2nd generation HMC, *HMC-15G-SR* [3]. A number of `PIMcore`s – which we implemented by extending ZSim's `SimpleCore` – build up the logic layer (figure 4). The number of `PIMcore`s is set equal to the number of vaults – vertical partitions of the 3-D stacked memory across the memory layers (figure 4) – in the DRAM. Hence, there is a `PIMcore` for each vault. The number of vaults, in turn, is determined by the total storage capacity (T), the bank size (B) and the number of layers (L) as $T/(B \times L)$ [3]. Fine-grained latencies inside the DRAM are modeled based on the HMC design [3] as well (table 3). Additionally, the architecture can be configured to add a data cache to each `PIMCore`, and an instruction cache shared by all `PIMCore`s (table 2).

Table 1: Host side processor cache configuration. All caches use LRU replacement policy.

Cache Type	Access Latency (CPU Cycles)	Size (KB) & associativity
L1-I	4	32, 4-way
L1-D	4	32, 4-way
L2	8	256, 4-way
L3	24	8192, 4-way

Table 2: Configuration of memory-side caches. All caches use LRU replacement policy.

Cache Type	Access Latency (CPU Cycles)	Size (KB) & associativity
`PIMCore-D`	4	1, Fully assoc.
`PIMCore-I`	4	32, 8-way

Table 3: Latencies inside the DRAM: RAS-to-CAS delay (t^{RCD}), row precharge time (t^{RP}) and CAS delay (t^{CL}). Please refer to [3] for more details.

Latency Type	Latency (CPU cycles)
t^{RCD}	47
t^{RP}	47
t^{CL}	47

We extended ZSim's `SimpleMemory` module to modeled basic functions of a 3-D stacked memory as well as more detailed fine-grained latencies internal to the DRAM. Although `PIMcore`s can access row buffers across the stacked data storage layers within their respective vaults with a high bandwidth through the TSVs, accesses to other vaults are slow due to inter-vault communication overheads [19]. Further, when a `PIMcore` is unable to find its request item in the row buffer (4 KB), it takes $t^{RP} + t^{RCD} + t^{CL}$ cycles to access that data item. Here, the RAS-to-CAS delay (t^{RCD}), row precharge time (t^{RP}) and CAS delay (t^{CL}) values are drawn from HMC specifications [3]. On the other hand, on a row-buffer "hit", it requires t^{CL} cycles to activate a column from an already activated row to access the data item. Also, application data is striped across memory banks to maximize throughput to `PIMcore`s in the logic layer.

We integrated these components to seamlessly execute user-defined PIM-functions in real user-space applications on the `PIMcore`s.

5.3 PIM Functions

To enable the execution of memory-bound functions on `PIMcore`s, we introduce a specialized function type called `PIM function`. `PIM function`s are the same as ordinary functions with the difference that the user may specify them to execute on `PIMCore`s. A `PIM function` completely executes on `PIMcore`s upon invocation, is packed as a dynamically loaded module, and gets linked at run time to the application. When an application issues a request to run a `PIM function`, the main processor stalls and transfers control to the `PIMCore`s through a special request to the memory. `PIMcore`s then divide and serve the request in parallel, utilizing the high bandwidth pathway available to them through the TSVs. The results of `PIM function`s may be returned to calling routine as a list of logical pointers to data items in memory.

5.4 End-to-End Workflow

We are able to use the architecture described earlier to:

1. Identify PIM functions in the user-program.
2. Coordinate the execution of PIM functions on PIM cores.
3. Model in-memory operations inside the `PIM DRAM` and the detailed PIM DRAM latencies for PIM cores to access memory.

When an application running on the main processor calls a function to perform operations using PIM, a DRAM line write request is made to a special address outside of the usable address space, and a portion of the execution context (described next) is written to the data segment of the application's virtual address space. When the DRAM controller observes a request to this address, it loads the program counter (PC), arguments passed to the function and other related execution context from the application's data segment in its virtual address space. Please note, this assumes that the address space layout is static (no address space layout randomization) and the CR3 register's value is known by – or can be communicated to – the DRAM controller to perform the necessary page table translations. The DRAM controller then sets up the `PIMcore`s to process this request by loading the required instructions from the instruction cache in the `PIM DRAM`. Assuming data striping, PIM cores can maximize their memory throughput by sending parallel requests across vaults. After `PIMCore`s complete processing, the DRAM controller sends an interrupt – specifically, a PIM request completion notification – to the main processor. The PIM-based application, which was likely context-switched out earlier, can wake up on this interrupt and start regular processing from the return address of the `PIM function`.

6 Evaluation

All benchmarks and evaluations were performed on a quad-core Linux "precise64" virtual machine with 4 GB of RAM.

6.1 Yahoo Cloud Service Benchmark (YCSB)

We used Yahoo's Cloud Service Benchmark (YCSB) [7] to evaluate the effectiveness of our PIM architecture. YCSB is a popular cloud benchmark used to assess the performance of key-value stores and databases. The benchmarking tool lets the user vary the load (total number of operations) and the kind of operations (e.g., insert, updates and deletes) performed, and can also be used to generate two benchmark distributions/patterns of key accesses which emulate distinct real-world workloads.

We implemented an example program using `libcuckoo` [24] to insert and query keys from a Cuckoo hashing-based key-value store. Finally, we generated a dataset with (i) uniform and (ii) Zipfian query distribution and 10K entries each, each with a 24-byte string as the key and a 1KB random string as the value. The Zipfian workload is useful in modeling several real-world applications, where there is *hot* data and *cold* data. Each slot in the hash table stores the key and the value (please note, as described in figure 2, the table is made up of buckets, with each bucket consisting of four slots). The table is initialized to 64 KB. The benchmark program executes 1 million point-queries on both datasets. Since our preliminary experiments demonstrated that batching requests effectively amortized the communication costs and other overheads related to `PIM DRAM`, we execute queries in batches of size 1024.

6.2 Concurrent Applications

In addition to Cuckoo hashing, we also implemented two auxiliary programs:

- *Large-scale array filtering:* This program filters elements in a large array (1 million elements), where each array element is a `struct` of two random integers, to find elements which lie inside a random range. The array is sorted on the first field of the `struct` and filtered on the second. Since this array requires 7.6 MB of storage space, it would take up almost all of the 8 MB L3 cache in our simulated machine. Hence, running this filtering program concurrently with other programs can enable us to study cache contention behavior.
- *Sequential array scan:* This program just scans a large array of `struct`s multiple times. Similar to the abovementioned program, the `struct`s comprise two elements.

6.3 Experiments

We carried out four significant experiments with varying configurations of the PIM-based architecture.

1. *Large-scale array filtering program executes concurrently with the sequential array scan program.* This experiment provides an estimate of the upper bound on the speedup achieved by offloading a large-scale filtering processing – significantly memory bound – to the `PIMCore`. Streaming-based systems and databases often experience workloads similar to large-scale filtering, wherein a large amount of data is scanned to select a few data items eventually. This experiment also provides insights into the effects of PIM on cache contention.
2. *YCSB workload executes concurrently with sequential array scan.* This experiment represents a more realistic workload and, similar to the first experiment, can help study cache interactions between the two programs.

Each of these experiments was executed with and without using PIM, and with a varying number of `PIMCore`s. This, in turn, varied the amount of bandwidth available to the `PIMCore`s, since each `PIMCore` can access data in its vault in parallel with other cores (table 4). We also varied `PIM DRAM`'s architecture and benchmarked performance with and without `PIMCore`s data caches.

Table 4: Experiment configurations: Number of `PIMCore`s and `PIM DRAM` size and layout. Varying the number of `PIMCore`s varies the amount of bandwidth available to the `PIMCore`s, since each `PIMCore` can access data in its vault in parallel with other cores (table 4)

# `PIMCore`s	`PIM DRAM` size	Bank size	# memory layers
16	16 GB	256 MB	4
32	32 GB	256 MB	4
64	64 GB	256 MB	4

7 Results

For each graph hereafter, we study the speedup achieved in terms of the total number of cycles executed in the main processor.

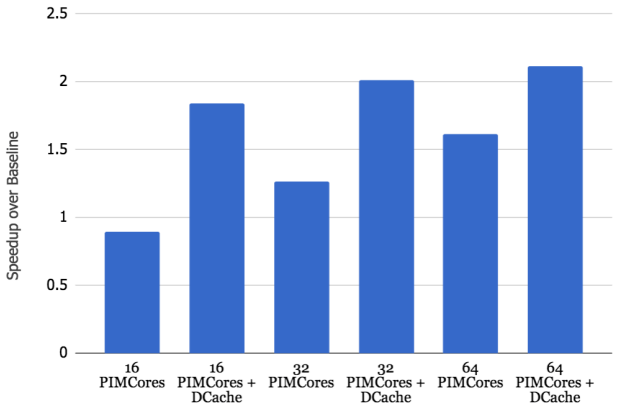

Figure 5: Speedup in large-scale array filtering when run concurrently with the sequential array scan program. The baseline is running both programs without PIM.

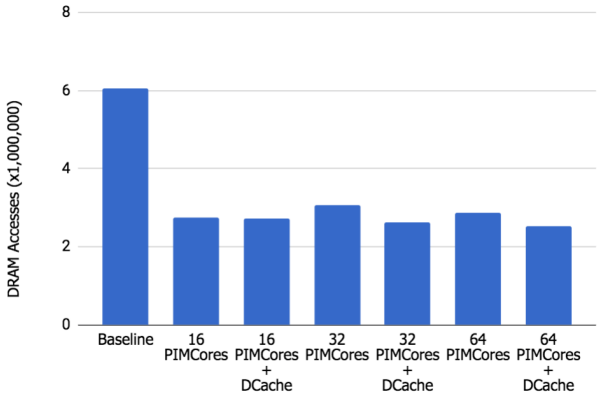

Figure 6: LLC misses for large-scale array filtering when run concurrently with the sequential array scan program.

As the number of `PIMCore`s is increased, the speedup in array filtering is observed to improve (figure 5). This is mainly due to the increased amount of parallelism made available by a greater number of vaults as the DRAM size increases. Since our design of the `PIM DRAM` includes a `PIMCore` under every vault, it is beneficial to increase the number of vaults for memory-bound applications. When data caches (one per `PIMCore`) are enabled, cores can access the rows from their vaults' banks at L1 latencies (for hits). Since array scan/filtering programs enjoy spatial locality, this greatly reduces the overhead induced by column activations which would otherwise be required even on DRAM row-buffer hits. We can also provide an upper bound on the speedup obtained by using PIM if we consider that all accesses from `PIMCore`s are hits. In this case, each access has a latency equal to L1 latency (four cycles in our case). Therefore, for 64 `PIMCore`s, the minimum total number of cycles required to execute the program using PIM is $4 \times$ len_array $= 4 \times 1,000,000 = 4,000,000$. We observed the number of cycles required using only the main processor (no PIM) to be $1,564,316,373$. Therefore, we expect our speedups to be less than the upper bound of $1,564,316,373/($ num_cores $\times 4,000,000) = 6.11$.

For an appropriate number of `PIMCore`s, the PIM architecture is observed also to be useful in speeding up Cuckoo hashing, another workload where a significant fraction of the work is memory operations.

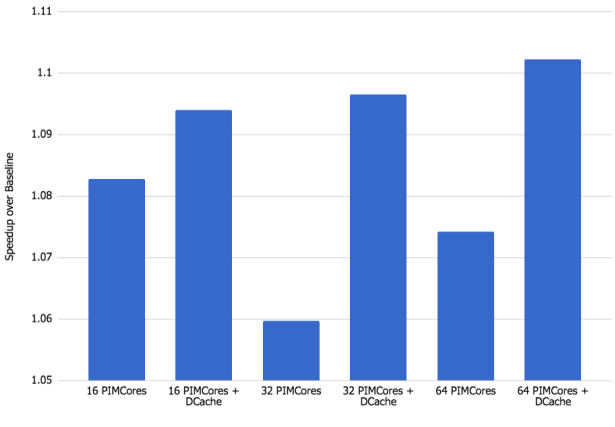

Figure 7: Speedup of sequential array scan – which does not use PIM functionality – running along with PIM-based array filtering. The baseline is running both programs without PIM.

We also observe (figure 7) that utilizing PIM results in a speedup in the sequential array scan program – which does not use PIM functionality – running along with PIM-based array filtering. Since data accesses for array filtering are offloaded to the PIM, the filtering program interferes with the cache requirement of the sequential scan to a lesser extent than it would have without PIM. Hence, PIM can also improve the performance of applications which do not use PIM operations at all by virtue of reduced cache contention.

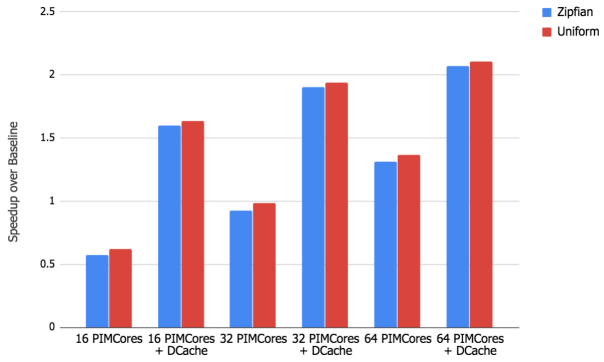

Figure 8: Speedup of YCSB workload when running concurrently with the sequential array scan program. The baseline is running both programs without PIM.

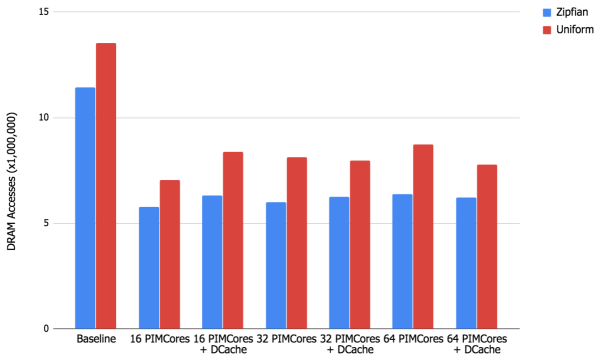

Figure 9: LLC misses for YCSB workload when running concurrently with the sequential array scan program.

Finally, we observe a significant reducing the amount of data transferred between the main processor and the DRAM on the processor-DRAM bus (figures 6 and 9). Since this bus is a significant consumer of energy in contemporary systems, PIM can help provide useful energy savings for energy-constrained or energy-efficient systems. Even though there is extra energy consumption by additional logic on the DRAM, we believe that this will be offset by the energy gains observed from reducing the processor-DRAM interconnect usage.

8 Limitations

1. Presently, computationally expensive hash computations required to execute key-value lookups are performed on wimpy `PIMCore`s.
2. We assume that the values in the hash table do not change during the lookups, i.e., we do not support coherence between the `PIMCore`s and the main caches.
3. ZSim does not support address (virtual to physical) translations or page tables. Performing DRAM modeling by solely using virtual addresses is not representative of real systems.
4. We do not model fine-grained latencies for the PIM controller for PIM request scheduling tasks.

9 Future Work

1. One direction of future work we wish to explore is to modify internal `libcuckoo` routines to compute hash values on the main processor and then transfer the value – as an additional argument – to the `PIMCore`s.

2. Another direction is the addition of fine-grained latency modeling for PIM task-scheduling.

3. A final direction we are interested in exploring is benchmarking applications on real hardware, such as AMD's High Bandwidth Memory enabled systems [1].

10 Conclusions

We briefly explored various design choices that a PIM architect could face in designing a PIM-based system for specialized workloads and the types of applications and workloads which can benefit from PIM.

Firstly, PIM is very effective when multiple applications contend for caches. In these scenarios, it is helpful to offload memory intensive operations to the PIM module.

Secondly, PIM – by helping reduce invocations of the processor-DRAM bus, a significant energy consumer in contemporary systems – can prove to be a useful option for energy-constrained or energy-efficient systems.

We believe several future developments and design iterations may be required to address issues related to programmability, generality, and coherence before PIM-based architectures gain widespread commercial acceptance.

11 Additional Notes

1. Some text for this report is drawn from the project proposal and midway report we submitted earlier.

2. Division of work: Equal work was performed by all project members.

3. We were able to achieve the 100% goal – studying the performance of cuckoo hashing and array filtering benchmarks with in-memory filtering operations – specified in our proposal.

12 References

[1] High bandwidth memory. *https://www.amd.com/en/technologies/hbm*.

[2] Micron's hybrid memory cube (hmc). *https://www.amd.com/en/technologies/hbm*.

[3] Micron's hybrid memory cube (hmc). *https://www.micron.com/resource-details/e0c00145-b43b-462c-9758-cae4559ce71f*.

[4] Boroumand, A., Ghose, S., Patel, M., Hassan, H., Lucia, B., Hajinazar, N., Hsieh, K., Malladi, K. T., Zheng, H., and Mutlu, O. Lazypim: Efficient support for cache coherence in processing-in-memory architectures. *arXiv preprint arXiv:1706.03162* (2017).

[5] Brockman, J. B., Kogge, P. M., Sterling, T. L., Freeh, V. W., and Kuntz, S. K. Microservers: A new memory semantics for massively parallel computing. In *Proceedings of the 13th international conference on Supercomputing* (1999), ACM, pp. 454–463.

[6] Brockman, J. B., Thoziyoor, S., Kuntz, S. K., and Kogge, P. M. A low cost, multithreaded processing-in-memory system. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture* (2004), ACM, pp. 16–22.

[7] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. *https://github.com/brianfrankcooper/YCSB*.

[8] Draper, J., Chame, J., Hall, M., Steele, C., Barrett, T., LaCoss, J., Granacki, J., Shin, J., Chen, C., Kang, C. W., et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing* (2002), ACM, pp. 14–25.

[9] Elliott, D. G., Stumm, M., Snelgrove, W. M., Cojocaru, C., and McKenzie, R. Computational ram: Implementing processors in memory. *IEEE Design & Test of Computers 16*, 1 (1999), 32–41.

[10] Fan, B., Andersen, D. G., and Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI* (2013), vol. 13, pp. 371–384.

[11] Ghose, S., Hsieh, K., Boroumand, A., Ausavarungnirun, R., and Mutlu, O. Enabling the adoption of processing-in-memory: Challenges, mechanisms, future research directions. *arXiv preprint arXiv:1802.00320* (2018).

[12] Gokhale, M., Holmes, B., and Iobst, K. Processing in memory: The terasys massively parallel pim array. *Computer 28*, 4 (1995), 23–31.

[13] Hall, M., Kogge, P., Koller, J., Diniz, P., Chame, J., Draper, J., LaCoss, J., Granacki, J., Brockman, J., Srivastava, A., et al. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing* (1999), ACM, p. 57.

[14] Hsieh, K., Khan, S., Vijaykumar, N., Chang, K. K., Boroumand, A., Ghose, S., and Mutlu, O. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on* (2016), IEEE, pp. 25–32.

[15] Hughey, R. Parallel hardware for sequence comparison and alignment. *Bioinformatics 12*, 6 (1996), 473–479.

[16] Iobst, K. W., Resnick, D. R., and Wallgren, K. R. Reconfigurable memory processor, Mar. 7 1995. US Patent 5,396,641.

[17] Ito, N., Sato, M., Kuno, E., and Rokusawa, K. *The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D*, vol. 14. IEEE Computer Society Press, 1986.

[18] Jasionowski, B. J., Lay, M. K., and Margala, M. A processor-in-memory architecture for multimedia compression. *IEEE transactions on very large scale integration (VLSI) systems 15*, 4 (2007), 478–483.

[19] Kim, J. S., Senol, D., Xin, H., Lee, D., Ghose, S., Alser, M., Hassan, H., Ergin, O., Alkan, C., and Mutlu, O. Grim-filter: fast seed filtering in read mapping using emerging memory technologies. *arXiv preprint arXiv:1708.04329* (2017).

[20] Kim, Y., Yang, W., and Mutlu, O. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters 15*, 1 (2016), 45–49.

[21] Kogge, P. M., Bass, S. C., Brockman, J. B., Chen, D. Z., and Sha, E. Pursuing a petaflop: Point designs for 100 tf computers using pim technologies. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the* (1996), IEEE, pp. 88–97.

[22] Kogge, P. M., Brockman, J. B., Sterling, T., and Gao, G. Processing in memory: Chips to petaflops. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA* (1997), vol. 97, Citeseer.

[23] Kogge, P. M., Sunaga, T., Miyataka, H., Kitamura, K., and Retter, E. Combined dram and logic chip for massively parallel systems. In *Advanced Research in VLSI, 1995. Proceedings., Sixteenth Conference on* (1995), IEEE, pp. 4–16.

[24] Li, X., Andersen, D. G., Kaminsky, M., and Freedman, M. J. Fast concurrent cuckoo hashing. *https://github.com/efficient/libcuckoo*.

[25] Liu, C. C., Ganusov, I., Burtscher, M., and Tiwari, S. Bridging the processor-memory performance gap with 3d ic technology. *IEEE Design & Test of Computers 22*, 6 (2005), 556–564.

[26] Loh, G. H. 3d-stacked memory architectures for multi-core processors. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on* (2008), IEEE, pp. 453–464.

[27] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices* (2005), vol. 40, ACM, pp. 190–200.

[28] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[29] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. A case for intelligent ram. *IEEE micro 17*, 2 (1997), 34–44.

[30] Pfaff, B., and Davie, B. The open vswitch database management protocol.

[31] Sanchez, D., and Kozyrakis, C. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news* (2013), vol. 41, ACM, pp. 475–486.

[32] Seshadri, V., Kim, Y., Fallin, C., Lee, D., Ausavarungnirun, R., Pekhimenko, G., Luo, Y., Mutlu, O., Gibbons, P. B., Kozuch, M. A., et al. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), ACM, pp. 185–197.

[33] Seshadri, V., Lee, D., Mullins, T., Hassan, H., Boroumand, A., Kim, J., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (2017), ACM, pp. 273–287.

[34] Upchurch, E., Sterling, T., and Brockman, J. Analysis and modeling of advanced pim architecture design tradeoffs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (2004), IEEE Computer Society, p. 12.

[35] Wilkes, M. V. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News 29*, 1 (2001), 2–7.

[36] Woo, D. H., Seong, N. H., Lewis, D. L., and Lee, H.-H. S. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010), IEEE, pp. 1–12.

[37] Zhang, D. P., Jayasena, N., Lyashevsky, A., Greathouse, J., Meswani, M., Nutter, M., and Ignatowski, M. A new perspective on processing-in-memory architecture design. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (2013), ACM, p. 7.