Architecting HBM as a High Bandwidth, High Capacity, Self-Managed Last-Level Cache

Tyler Stocksdale[1], Mu-Tien Chang[2], Hongzhong Zheng[2], Frank Mueller[1]

[1]North Carolina State University, Raleigh, NC
[2]Samsung Semiconductor Inc., San Jose, CA

ABSTRACT

Due to the recent growth in the number of on-chip cores available in today's multi-core processors, there is an increased demand for memory bandwidth and capacity. However, off-chip DRAM is not scaling at the rate necessary for the growth in number of on-chip cores. Stacked DRAM last-level caches have been proposed to alleviate these bandwidth constraints, however, many of these ideas are not practical for real systems, or may not take advantage of the features available in today's stacked DRAM variants.

In this paper, we design a last-level, stacked DRAM cache that is practical for real-world systems and takes advantage of High Bandwidth Memory (HBM) [1]. Our HBM cache only requires one minor change to existing memory controllers to support communication. It uses HBM's built-in logic die to handle tag storage and lookups. We also introduce novel tag/data storage that enables faster lookups, associativity, and more capacity than previous designs.

1 INTRODUCTION

Commodity DRAM is hitting the memory/bandwidth wall. Caching has traditionally avoided this wall by alleviating the pressure on off-chip memory. Cache is not only faster than off-chip memory, but reduces the number of requests going to DRAM. However, modern workloads are demanding hundreds of megabytes of last level cache (LLC) [3, 4].

This is a problem because there is a large capacity gap between existing LLC's and off-chip memory. The capacity gap can be to four orders of magnitude, while bandwidth and latency can see a gap of up to one order of magnitude. Many different ideas have been proposed in an effort to close these gaps. Simply architecting a larger, traditional SRAM cache is the most trivial solution. However, SRAM caches are low density, power hungry, and costly. Another option is to construct LLC's using DRAM since it is higher density, less power hungry, and low cost. Despite these benefits, the relatively long latency of conventional DRAM prevents it from being an effective cache. Even other technologies such as eDRAM [6] and STT-MRAM [7] are infeasible due to their limited capacity. However, 3D stacked DRAM has been shown to provide more bandwidth and less latency than conventional off-chip DRAM [5], while maintaining a capacity that is acceptable for LLC's in modern workloads. Effectively, stacked DRAM is able to close the bandwidth, latency, and capacity gaps currently exhibited between existing LLC's and off-chip memory, all while maintaining acceptable density levels, power consumption, and cost. For this reason, stacked DRAM LLC's have emerged as the best proposed solution to this problem, the best of which have shown up to 21% system performance improvement [2].

However, many of these proposed solutions are not practical for real systems, requiring major changes to existing hardware and DRAM standards. In addition, the proposed solutions may not take advantage of the many features available in today's stacked DRAM variants. To that end, this paper makes the following contributions:

(1) We design a stacked DRAM LLC using HBM [1]. This cache is practical for real systems and takes advantage of the available logic die in HBM to accomplish cache management, tag storage, and tag lookups in memory.
(2) We introduce a novel way to store the cache tags and data inside the stacked DRAM. Our method uses sub-array level parallelism (SALP) [8] in order to achieve fully concurrent tag and data accesses. This promotes faster lookups, enables associativity, and increases the usable capacity compared to prior work.

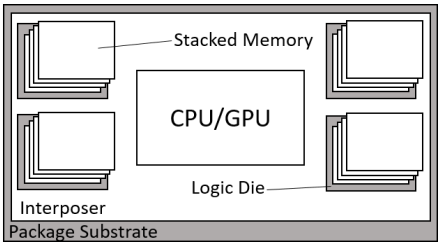

Figure 1: Multiple HBM stacks on the interposer [10]

PDSW-DISCS'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5134-8/17/11…$15.00
https://doi.org/10.1145/3149393.3149394

2 BACKGROUND

HBM is designed as a series of stacked DRAM dies which sit atop a base-layer logic die that can be customized and used for processing in memory (PIM). A visual representation of this architecture can be seen in Figure 1.

Each HBM stack contains 8 completely independent memory channels. Each channel has a 128-bit data interface that provides 32GB/s bandwidth. This means that, given a system with 4 HBM stacks, each with 8 channels, the maximum theoretical bandwidth would be 1TB/s. The capacity of each HBM stack can range from 1 to 32 GB. Each channel is similar to a standard DDR interface and requires traditional DRAM controller command sequences. There are 16 banks per channel, but each channel also can be split into two "pseudo-channels" each with an independent 64-bit I/O line. Effectively, the pseudo-channel mode doubles the amount of channels, but halves the column width of each bank.

In this work, we use one HBM stack of 4 DRAM dies with a total capacity of 4GB and 16 pseudo-channels of 8 banks each for a maximum theoretical bandwidth of 256GB/s.

3 RELATED WORK

There are many existing proposals for stacked DRAM LLC's [2, 9, 11, 12] but none of these use processing in memory, which is available by using HBM's built-in logic die. In most prior work, tag and data access takes two memory transactions from the memory controller. Additionally, the memory controller must be modified to perform tag comparisons and implement all other cache management logic.

Cache Tag Storage: As the size of a cache grows, so does the size of the tags and other metadata needed for it to function. In practice, the most popular, practical, and often simplest solution is to store the tags in the stacked DRAM [13]so that access to the tag and data can happen serially or in parallel. A serial access has long latency, but ensures that data is accessed only if necessary. A parallel access saves latency, but since the tag and data access happen at the same time, there are some situations where the data access was unnecessary and wasteful. When optimizing for performance, the parallel access scheme is preferred.

Alloy Cache: The Alloy Cache [2] is a latency-optimized cache architecture that is currently seen as the "best candidate for an off-chip Giga-scale DRAM cache" [11] with a 21% performance increase over a baseline that does not have a stacked DRAM cache. It is implemented as the sole DRAM cache model in the NVMain simulator and used as an aggressive baseline in multiple studies [11, 12]. For these reasons it is also the baseline for our work.

The Alloy Cache is designed to reduce hit latency for stacked DRAM caches. To this end, the Alloy Cache is direct mapped, eliminating the latency incurred by multiple tag comparisons and replacement policies. Additionally, cache tags are "alloyed" with the corresponding data in the stacked DRAM to create a tag-and-data (TAD) unit. One TAD unit is streamed out on every cache access, which avoids a time-consuming serial tag/data access and simplifies the wasteful parallel tag/data access.

The 72 byte TAD (8B tag and 64B data) takes 5 bus cycles to transfer as opposed to the standard 4 cycles for DRAM. Not only does this require a major change to existing memory controller designs and DRAM standards, but due to the unusual size, it sacrifices capacity in the DRAM cache (Figure 2). For a Giga-scale DRAM cache, this amounts to tens of megabytes being wasted. Also, the experimental configuration for the Alloy Cache study does not seem to specifically model any of the existing stacked DRAM variants.

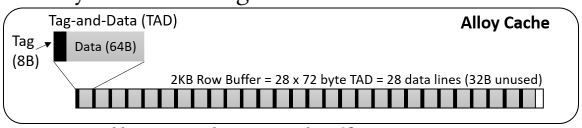

Figure 2: Alloy Cache row-buffer organization using tag-and-data (TAD) units [2]

Intel's Knights Landing processor [9] uses a 16 GB MCDRAM LLC with a reported 500 GB/s bandwidth. The LLC is known to be direct mapped, but other specifics of the design are not detailed, i.e., additional cache management strategies and tag/data organization are unknown.

4 HBM CACHE DESIGN

The design of our HBM cache is similar to a traditional cache. It is direct mapped with a write-back, write allocate policy, and a cache block size of 64B with tags to uniquely identify blocks of data. However, our design differs in two novel ways. First, tag operations (including comparisons, lookups, and storage), address/command translations, and all other cache management is done in-memory on the HBM logic die, resulting in a self-managed cache. Second, our design uses a novel way of storing tags and data within the stacked DRAM. Finally, we assume the HBM cache is accessed via a single memory controller (i.e., no NUMA support).

4.1 Self-Managed Cache

The HBM spec provides for a logic die, which we use to accomplish cache management in-memory, without the need for a specialized memory controller. This means that the system memory controller can treat our HBM cache as an ordinary DRAM device, using all of the ordinary DRAM commands. There is no need for tag comparisons, etc. at the memory controller level. With this design, we can guarantee synchronous operations, just like the HBM spec, but it may require extra cycles for all the logic and buffer operations. Since the HBM spec does not specify certain timing parameters, we can extend traditional DRAM timing by a few cycles.

The memory controller needs to be modified so that an access to the HBM cache results in either a hit or a miss. Depending on the outcome, the memory controller needs to react in a different way. Namely, it needs to decide if a further access to the system memory is necessary and what to do with any data returned from the HBM cache. E.g., in the case of a read hit, there is no further memory access necessary and the returned data is valid. In case of a write miss, the returned data is dirty and must be written to system memory.

In order to make this decision, we modify the memory controller to receive a "cache result signal".This is a one-bit signal sent over HBM's reserved for future use (RFU) pin. The signal is sent over multiple cycles and aligned with the data burst, allowing for multiple different bit combinations to be sent. Each combination corresponds to different cache results, which include, but are not limited to, hit, clean miss, dirty miss, invalid, etc. Using the standard burst length of 4 for HBM, it is possible to send up to 16 unique cache result signals. Possible other signals could include coherence information for multi-node systems.

This self-managed cache has important advantages. First, performing the cache management in-memory reduces the total time for each memory access compared to a system that performs cache management within the memory controller. This performance gain is due to a reduction in the amount of data traveling over the memory bus and a reduction in the amount of communication going off-chip. Second, there is no need for major changes to existing memory controllers.

Figure 3 depicts the duties of the logic die. E.g., when HBM needs to write some data, a write command and address is put on the CA bus, and the data is put on the DQ bus. First, the data is put into the Data Buffer. The write command and address is translated by the Command Translator and the Address Translator into a read command for the corresponding tag and a read command for the corresponding data. These two commands are sent to the scheduler and the Address Translator puts the expected tag into the Tag Comparator. The tag is fetched from the Stacked DRAM and is also put into the Tag Comparator while the data is fetched and put into the Data Buffer. The Tag Comparator compares the tags. If the tags match, a cache hit signal is sent back to the memory controller using the RFU pin, and the data in the Data Buffer is written to the Stacked DRAM. If the tags do not match, there are two possibilities:

- Invalid/Clean Miss: The tag indicates that the cache line is invalid, or that it is valid, but clean. An invalid/clean-miss signal is sent back to the memory controller using the RFU pin, the data in the Data Buffer and the new tag are written to the Stacked DRAM.
- Dirty Miss: The tag indicates that the cache line is valid and dirty. A dirty-miss signal is sent back to the memory controller using the RFU pin and the dirty data in the Data Buffer is sent back on the DQ bus. The new data in the Data Buffer and the new tag are written to the Stacked DRAM.

These steps implement a traditional caching algorithm, yet with the notable difference that a single DRAM command from the memory controller can result in multiple DRAM commands for the Stacked DRAM, generated by the Command Translator and Address Translator. In the above data write example, one DRAM command from the memory controller could result in up to four DRAM commands for the Stacked DRAM.

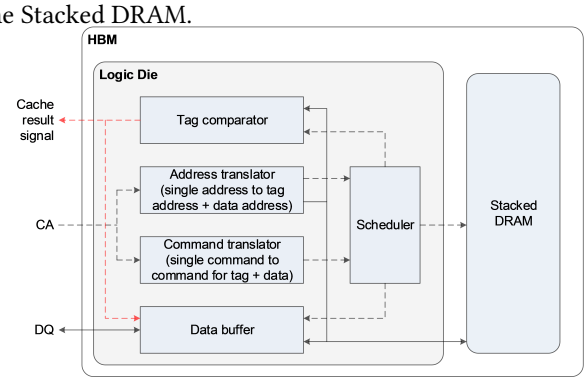

Figure 3: Logic die with in-HBM cache manager

4.2 Tag and Data Storage

Unlike the Alloy cache where tags are alloyed with the data, we choose to store tags separately from the data. Our design reserves one pseudo-channel for tag storage, and the other 15 pseudo-channels for data storage. This enables parallel access to the tags and data and also wastes less capacity than the Alloy cache. Accessing the tags and data in parallel means that there are unnecessary and wasteful data accesses, but since our HBM cache is self managed, these unnecessary accesses are internal to HBM, are not using the memory bus, and are therefore not as wasteful. Recall that the Alloy cache wastes 32 bytes per row of DRAM for a total of 64MB of wasted cache capacity. By storing the tags separately from the data, we are able to completely fill the DRAM rows storing data without wastage here.

However, our design does waste a small amount of space in its tag storage. Given a total cache capacity of 4GB, each pseudo-channel can hold 256MB of tags or data. With 15 pseudo-channels for data storage and a 64B cache line size, we have 60M cache lines, and need to store 60M tags. With 256MB of tag storage, we have enough space for tags to be 4B (which is much greater than the 17 bits required for a 15 bit tag, a valid bit, and a dirty bit). However, storing 60M, 4B tags only requires 240MB, wasting a total of 16MB cache capacity. In comparison, our tag/data organization can store 4.2 million more cache lines than the Alloy cache TAD organization, while avoiding the elongated burst length of the Alloy cache.

One additional problem arises in our tag/data storage solution. Since tags are 16x smaller than the data they correspond to, accessing data in parallel may cause bank conflicts for the corresponding tags. Figure 4 shows that Data H0 and Data H1 can be accessed in parallel, but their respective tags are stored in the same bank, causing a bank conflict and requiring a serial bank access. The solution to this is to use subarray level parallelism (SALP) [8]. With SALP, each bank is divided into 16 subarrays, which can be accessed in parallel. Each tag is stored in a different subarray, thereby avoiding a serial access due to bank conflicts and enabling full parallelism.

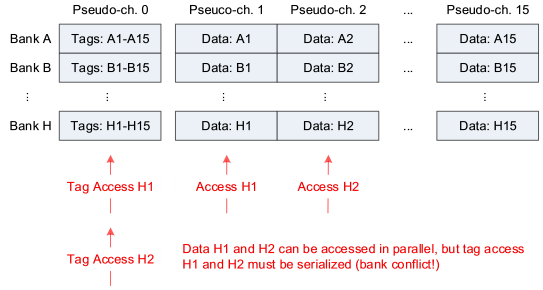

Figure 4: Without SALP, a bank conflict can occur

4.3 Cache Configurations

Given our two modifications to a traditional cache, we propose three cache configurations for evaluation: (1, "Alloy") A traditional host-managed Alloy cache, our baseline, that does not use HBM's logic die and will have the tag/data stored as a TAD; (2, "Alloy-like") a self-managed cache, that still uses the TAD style tag/data storage; (3, "SALP") a cache, also self-managed, but uses SALP to enable our novel tag/data storage organization. Figure 5 highlights the differences per configuration. With (1), the memory controller must handle both the tag and the data. With the self-managed style of (2) and (3), the tag is local to the logic die and DRAM stack and never crosses over the memory bus to the memory controller. In (2), a TAD unit is still transferred between the logic die and DRAM stack, while (3) replaces this with a parallel tag/data access. The self-managed style of (2) and (3) also introduces the cache result signal.

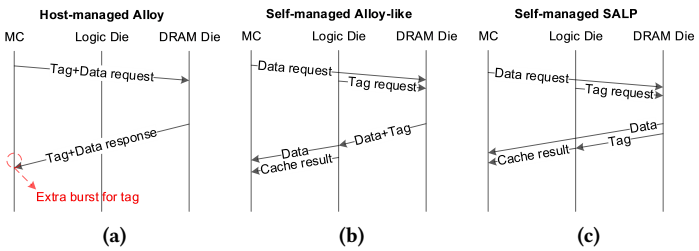

Figure 5: Different cache configurations

4.4 Theoretical Results

For these configurations, one can derive theoretical performance limits. The theoretical capacity, bandwidth, and latency values for each cache configuration are depicted in Figures 6, 7, and 8, respectively. The effective cache capacity corresponds to usable capacity for storing data, excluding metadata. The theoretical bandwidth corresponds to bandwidth for data, excluding metadata. In Figure 6, the effective capacity of SALP is greater than the Alloy and Alloy-like because of our novel tag/data storage organization. The total capacity of our HBM stack is 4GB. In Figure 7, the bandwidth of SALP is increased because of the elimination of possible bank conflicts through the use of SALP. The maximum possible bandwidth of our HBM stack is 256GB/s. In Figure 8, the latency of a read hit is slightly reduced in SALP due to the shorter data burst length. The latency of a write hit is reduced significantly in Alloy-like and SALP configurations because of the self-managed cache. Additionally, the write hit latency is slightly lower in SALP due to the shorter data burst length. These latency values were calculated based on Samsung's DDR4 8GB spec to estimate HBM timing parameters.

5 EXPERIMENTAL METHODS

Simulator Configuration: In order to evaluate and test our design ideas, we used the GEM5 [14] simulation software integrated with NVMain [15].

Each of these simulators can be configured with parameters such as:

- Number of CPUs, frequency, bus width, bus frequency;
- Cache size, associativity, hit latency, frequency;
- DRAM timing parameters, architecture (banks, ranks, channels, rows, cols), energy/power parameters.

These parameters are easily modified to support rapid hardware re-configuration and evaluation. We used projected HBM timing numbers based on DRAMSpec [16] and JEDEC DDR4 [17]. We project PCM timing numbers based on work done by Tschirhart [18]. Figure 9 gives a system overview and Table 1 shows the configuration used in our study.

Workloads: We used multiple benchmarks from the PARSEC [19] and NAS [20] benchmark suites, each of which was simulated for at least 1 billion instructions. The benchmarks were selected to have a variety of memory access patterns.

6 RESULTS

Our simulations compare our self-managed HBM cache to the baseline Alloy Cache with regard to bandwidth and execution time.

Figures 10 and 11 depict the performance benefit (y-axis) in terms of percentage change (higher is better) per benchmark (x-axis). Some benchmarks perform very well under our design (dedup, ft, mg), and a large number exhibit similar performance. Some benchmarks (e.g., streamcluster, fluidanimate) performed worse with our design. These are not included in the reported results because in future work, we plan to dynamically select SALP only when benchmarks can be predicted to benefit from our cache configuration.

It is worth noting that these results were obtained using a close-page memory policy along with a coarse-grained (bank

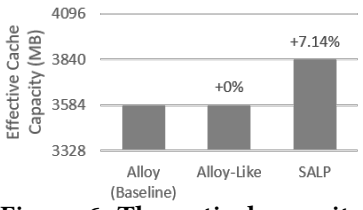

Figure 6: Theoretical capacity for a 4GB array

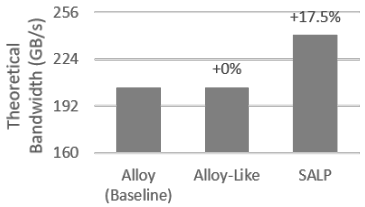

Figure 7: Theoretical bandwidth

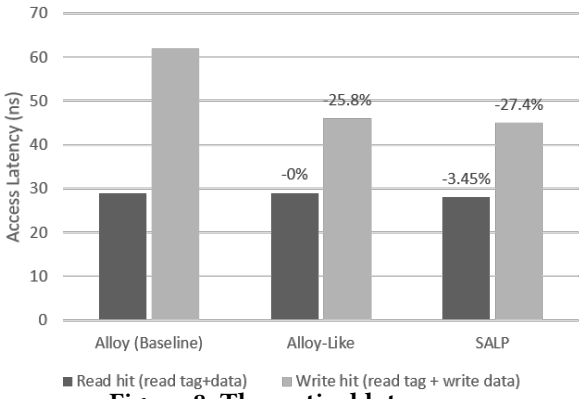

Figure 8: Theoretical latency

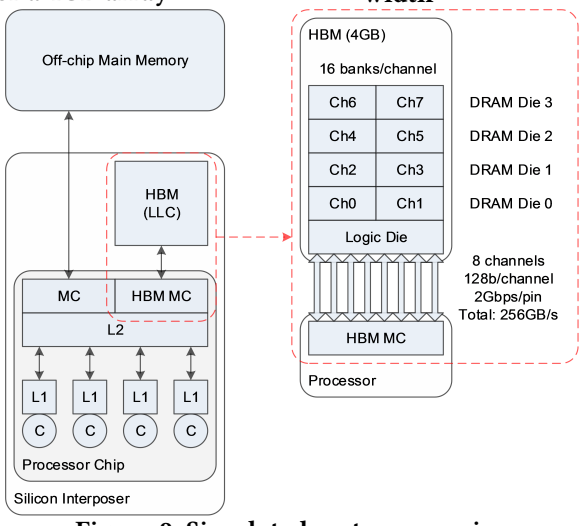

Figure 9: Simulated system overview

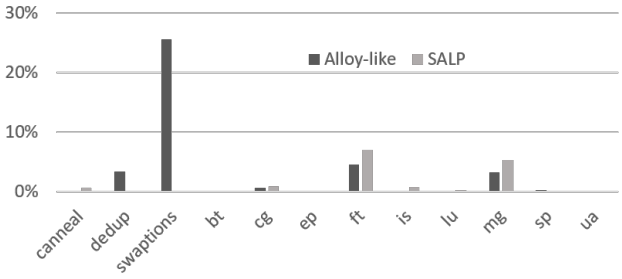

Figure 10: Bandwidth Performance Benefit

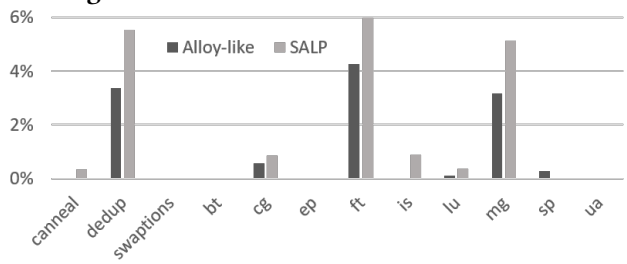

Figure 11: Execution Time Performance Benefit

Table 1: Simulator Configuration

Processors	
Number of cores	4
Frequency	4.0 GHz
Caches	
L1 (private)	32 KB, 8-way, 4 cycles
L2 (shared)	8 MB, 16-way, 44 cycles
Stacked DRAM LLC	4 GB, direct mapped
Stacked DRAM	
Bus frequency	800 MHz (DDR 1.6 GHz)
Pseudo-Channels	16
Ranks	1 per channel
Banks	8 per rank
Row buffer size	2048 bytes
Bus width	64 bits per channel
Off-Chip Memory (PCM)	
Bus frequency	800 MHz (DDR 1.6 GHz)
Channels	1
Ranks	2 per channel
Banks	8 per rank
Row buffer size	2048 bytes
Bus width	64 bits per channel

first) address mapping scheme. Six different combinations were tested: coarse-grained (channel first), coarse-grained (bank first), and fine-grained, each with both open-page and close-page. None of the other configurations performed better than the reported results in Table 2.

Table 2: Performance Relative to the Alloy Cache

	Bandwidth		Execution Time	
	Alloy-like	SALP	Alloy-like	SALP
Minimum	-0.30% (UA)	-0.72% (Dedup)	-0.20% (IS)	-0.42% (UA)
Maximum	25.53% (Swaptions)	7.07% (FT)	4.26% (FT)	6.59% (FT)
Arithmetic Mean	3.10%	1.22%	0.92%	1.73%
Geometric Mean	2.89%	1.19%	0.93%	1.76%

7 CONCLUSIONS

Based on our results, using HBM as a self-managed last-level cache can be beneficial in certain cases. Since our theoretical results indicate that there should be a tangible performance benefit in the best case (streaming access pattern), working towards categorizing benchmarks that will perform well with an HBM cache is our primary future goal. From this, a benchmark analysis could be performed to dynamically decide the optimal cache configuration and then allow ad-hoc selection of the best policy, which is particularly attractive in software managed HBMs (e.g., for Intel's KNL).

REFERENCES

[1] JEDEC Standard, "High Bandwidth Memory (HBM) DRAM," in *JESD235A*, 2015.

[2] M. K. Qureshi and G. H. Loh, "Fundamental latency tradeoff in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design", *International Symposium on Microarchitecture*, 2012, pp. 235-246.

[3] D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsari et al., "Thermal characterization of cloud workloads on a power-efficient server-on-chip", *International Conference on Computer Design (ICCD)*, 2012, pp. 175-182.

[4] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-stacked and Off-package Memories", *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[5] S. Mittal, and J.S. Vetter, "A Survey Of Techniques for Architecting DRAM Caches", *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[6] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd, "Power7: IBM's Next-Generation Server Processor", *IEEE Micro*, 2010, vol. 30, no. 2, pp. 7-15.

[7] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM", *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[8] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (SALP) in DRAM", *International Symposium on Computer Architecture (ISCA)*, 2012, pp. 368-379.

[9] (2014). [Online]. Available: http://wccftech.com/intel-xeon-phiknights-landing-processors-stacked-dram-hmc-16gb/

[10] (2015). [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/hbm

[11] B. Pourshirazi and Z. Zhu, "Refree: A Refresh-Free Hybrid DRAM/PCM Main Memory System", *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 566-575.

[12] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth", *International Symposium on Microarchitecture (MICRO)*, 2014, pp. 38-50.

[13] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms", *International Conference on Computer Design (ICCD)*, 2007, pp. 55-62.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator", *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1-7, 2011.

[15] M. Poremba, T. Zhang, and Y. Xie, "NVMain 2.0: Architectural Simulator to Model (Non-)Volatile Memory Systems", *Computer Architecture Letters (CAL)*, 2015.

[16] O. Naji, A. Hansson, C. Weis, M. Jung, N. Wehn, "A High-Level DRAM Timing, Power and Area Exploration Tool", *IEEE International Conference on Embedded Computer Systems Architectures Modeling and Simulation (SAMOS)*, 2015.

[17] JEDEC Standard, "DDR4 SDRAM Standard," in *JESD79-4A*, 2013.

[18] P. K. Tschirhart, "Multi-Level Main Memory Systems: Technology Choices, Design Considerations, and Trade-off Analysis.", 2015.

[19] C. Bienia, K. Sanjeev, J.P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications", *Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72-81.

[20] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks", *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63-73, 1991.