Monolithically Integrating Non-Volatile Main Memory over the Last-Level Cache

CANDACE WALDEN, DEVESH SINGH, MEENATCHI JAGASIVAMANI, SHANG LI, and LUYI KANG, University of Maryland, College Park
MEHDI ASNAASHARI and SYLVAIN DUBOIS, Crossbar Inc.
BRUCE JACOB and DONALD YEUNG, University of Maryland, College Park

Many emerging non-volatile memories are compatible with CMOS logic, potentially enabling their integration into a CPU's die. This article investigates such monolithically integrated CPU–main memory chips. We exploit non-volatile memories employing 3D crosspoint subarrays, such as resistive RAM (ReRAM), and integrate them over the CPU's last-level cache (LLC). The regular structure of cache arrays enables co-design of the LLC and ReRAM main memory for area efficiency. We also develop a streamlined LLC/main memory interface that employs a single shared internal interconnect for both the cache and main memory arrays, and uses a unified controller to service both LLC and main memory requests.

We apply our monolithic design ideas to a many-core CPU by integrating 3D ReRAM over each core's LLC slice. We find that co-design of the LLC and ReRAM saves 27% of the total LLC–main memory area at the expense of slight increases in delay and energy. The streamlined LLC/main memory interface saves an additional 12% in area. Our simulation results show monolithic integration of CPU and main memory improves performance by 5.3× and 1.7× over HBM2 DRAM for several graph and streaming kernels, respectively. It also reduces the memory system's energy by 6.0× and 1.7×, respectively. Moreover, we show that the area savings of co-design permits the CPU to have 23% more cores and main memory, and that streamlining the LLC/main memory interface incurs a small 4% performance penalty.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Computer systems organization → Multicore architectures;**

Additional Key Words and Phrases: Crosspoint architectures, ReRAM, on-die main memory systems

This work is an extension of a conference paper. Our previous work [23–25] explored the concept of monolithic integration and suggested the possibility of integrating ReRAM over the last-level cache. This article expands upon that with a detailed Cacti design study of a full cache slice including ReRAM access circuitry overheads as well as a simulation-based performance analysis of the proposed many-core CPU monolithically integrated with a ReRAM memory system. This represents an addition of 60% new material.

This work was supported by Semiconductor Research Corporation (contract 2020-AH-2937).

Authors' addresses: C. Walden, D. Singh, M. Jagasivamani, S. Li, L. Kang, B. Jacob, and D. Yeung, University of Maryland, College Park MD 20742, 1419 A.V. Williams Building; emails: cbwalden@umd.edu, dsingh19@terpmail.umd.edu, {mjagasiv, shangli}@umd.edu, luyikang@terpmail. umd.edu, blj@ece.umd.edu, yeung@umd.edu; M. Asnaashari and S. Dubois, 3200 Patrick Henry Dr., Suite 110, Santa Clara, CA 95054; emails: {mehdi.asnaashari, sylvain.dubois}@crossbar-inc.com.

1544-3566/2021/07-ART48 $15.00
https://doi.org/10.1145/3462632

ACM Reference format:
Candace Walden, Devesh Singh, Meenatchi Jagasivamani, Shang Li, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. 2021. Monolithically Integrating Non-Volatile Main Memory over the Last-Level Cache. *ACM Trans. Archit. Code Optim.* 18, 4, Article 48 (July 2021), 26 pages.
https://doi.org/10.1145/3462632

1 INTRODUCTION

In the post-Moore era, computer architects will no longer be able to rely on device scaling to improve the energy efficiency and performance of computer systems. Instead, new technologies will be needed to fuel the efficiency and performance gains of the future. A promising direction is to use emerging non-volatile memory technology to supplement or even replace DRAM in the memory system. Examples include **resistive RAM (ReRAM)**, **spin-transfer torque magnetic RAM (STT-MRAM)**, and **phase change memory (PCM)**.

Many researchers have already proposed main memory systems containing such non-volatile memories [1, 14, 16, 31, 40, 41, 46, 50]. These new memory systems provide much higher capacity at lower cost because the new non-volatile memories are denser and more scalable than DRAM. There is also a significant energy efficiency benefit for these new memory systems, in part because their non-volatility eliminates the need for refresh. One potential drawback of the new non-volatile memories, however, is that they can be slower than DRAM. In addition, writes are more costly than reads, not just in terms of latency but also in energy and endurance. Addressing these shortcomings has been a major focus of the current research in non-volatile main memory systems.

In this article, we explore another interesting characteristic of emerging non-volatile memories: their compatibility with CMOS logic. Whereas fabricating DRAM requires special VLSI processes tuned for implementing DRAM memory cells, fabricating non-volatile memory can be done within the context of a standard CMOS logic process. This implies that we can integrate non-volatile memory directly into the die of a CPU (or even a GPU). Compared to conventional discrete memory systems, such monolithically integrated memory provides less planar area to fabricate the combined compute and memory circuits. Nevertheless, thanks to its high density, significant amounts of non-volatile memory–potentially 100s of gigabytes—can still be integrated.

Monolithically integrating main memory into a CPU's die will improve computers in two major ways. First, it will enable placing the memory system physically close to the cores, virtually eliminating data movement for many memory accesses. This can significantly improve the energy efficiency. As shown in Figure 1, a request to board-level memory (DDR4) incurs 22 pJ/bit [32], whereas a request to **High Bandwidth Memory (HBM2)** incurs 5.9 pJ/bit. (The latter assumes 1.9 pJ/bit for traversing half the CPU die on average, and 4.0 pJ/bit [38] for energy on the silicon interposer and in the HBM stack.) If main memory were monolithically integrated into the CPU's die, we could reduce the distance separating a core and the accessed memory bank to as little as 1 mm. This would enable a memory request to incur just 2.5 pJ/bit. Of this, only 0.1 pJ/bit would be due to data movement thanks to the physical proximity, compared to 4.7 pJ/bit of data movement energy (out of 5.9 pJ/bit) for HBM2.

Second, monolithic integration will also enable a massively parallel connection from the cores to main memory. Depending on its organization, monolithically integrated non-volatile memory can have many thousands of independent memory banks. These memory banks can be distributed among the cores, affording *every core* the physical locality benefits mentioned earlier (at least to a portion of the main memory). Due to the high wire density on die, every core can also have a wide and dedicated connection to its local memory banks. This will result in large amounts of memory bandwidth despite the higher access latency of non-volatile memory compared to

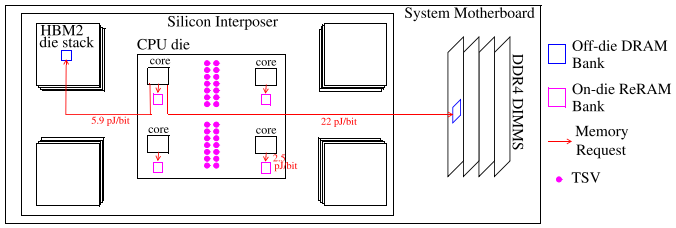

Fig. 1. Impact of physical distance and wire density on energy.

DRAM. We estimate that at a 200 ns access latency on the 14-nm technology node, an on-die memory system could support just over 1 TB/s. Given lower access latencies on more aggressive technology nodes, even higher memory bandwidths would become possible. In contrast, discrete memory systems, like DDR4 and HBM2 in Figure 1, are limited by a small number of I/O pins. Moreover, even if DRAM dies were stacked on top of the CPU, the interconnect density supported by the TSVs would still be one to two orders of magnitude lower than monolithic integration. Not only would this sacrifice memory parallelism, but it would also limit physical locality as the cores' memory requests would have to route to a smaller number of, and hence more distant, TSVs (e.g., along a middle stripe on the chip as shown in Figure 1), resulting in greater data movement energy.

Recently, researchers have begun investigating monolithically integrated CPU–main memory chips [3, 4, 23–25]. One problem addressed by the research in this area is that physically, the integration must permit an entire processor and memory system to fit into a single die. If the two are integrated in a 2D planar fashion, then the available area can become a limiting factor. But recent non-volatile memories allow for *monolithic 3D stacking* of the memory cells (as opposed to die stacking) to improve density. Examples include Intel's 3D XPoint [22] and Crossbar's 3D ReRAM [11]. In these 3D memories, the bitcells are sandwiched in between metal wires (i.e., at the intersection of wires laid out perpendicularly in adjacent VLSI layers) giving rise to a "crosspoint architecture." Rather than isolate individual bitcells using access transistors, isolation in crosspoint subarrays is provided via "selector devices."

The use of selector devices enables extremely small bitcells that can be stacked vertically across multiple metal layers. However, it also means the transistors underneath the crosspoint subarrays are free for implementing non-memory circuits. This implies the crosspoint memory can be fabricated over the CPU, occupying the top-level metal layers of the CPU's die. Meanwhile, the CPU's logic can be implemented in the die's logic transistors, minus those needed for the memory access circuits. Such placement of the memory system over the CPU can yield higher area efficiency.

Unfortunately, while the memory bitcells do not consume transistors, the non-volatile memory's access circuits (i.e., decoders and sense amplifiers) do. When integrated with random logic (e.g., the CPU's datapath), the access circuits can disrupt the compute circuits' layout, introducing significant area overheads [24, 25]. Alternatively, the crosspoint arrays can be integrated over cache. Prior work has observed that the regular structure of SRAM allows it to be placed underneath crosspoint arrays with minimal overheads [23, 24]. In this work, we exploit this basic observation and present the design of a 3D memory structure comprised of a complete **last-level cache (LLC)** slice and a crosspoint main memory module. Our design also places a coherence directory underneath the crosspoint memory as well.

In addition to physical integration, another problem is what is the best way to architect the LLC/main memory interface? On-die memory systems completely transform the interconnection between the LLC and main memory. The integration moves main memory onto the same die

as the **memory controller (MC)**, making the off-chip memory bus unnecessary. Moreover, by interspersing the memory system's subarrays among the LLC's cache mats, even the internal interconnects between the cache/memory controller and the cache mats/memory subarrays become redundant. In this article, we propose to provide a single shared internal interconnect that carries requests from both the cache controller to the cache mats as well as the MC to the memory subarrays. This shared interconnect further reduces the area impact of the integration. Not only are the internal interconnects of the LLC and memory system redundant, but so are the cache controller and MC. We further propose a single unified controller for both the cache and memory system. Our unified controller retains the cache's **miss status holding registers (MSHRs)**, but it integrates the MC's scheduler queue into the cache's MSHRs.

This work makes the following contributions:

- We integrate non-volatile main memory into the die of a highly parallel CPU—specifically, an in-order multi-threaded many-core with two levels of cache. Our work considers both a coherent and non-coherent version of the cache hierarchy. We associate a portion of main memory with each core, resulting in high physical locality and access parallelism.
- We study physically integrating crosspoint subarrays over the LLC. In particular, our study considers 3D ReRAM [11]. Using Cacti, we determine the best cache mat size for placement underneath the ReRAM, then optimize the layout of the integrated cache and ReRAM subarrays to form a complete LLC slice/main memory module. We also use leftover area to additionally integrate an L1 directory. Co-designing the LLC and main memory module saves 27% in area with manageable increases in delay and energy for LLC accesses.
- We explore sharing a single internal interconnect between the controller(s) and the cache mats and ReRAM subarrays. This provides an additional 12% area savings. We also propose a unified controller for servicing both LLC and main memory requests.
- Finally, we evaluate our monolithically integrated CPU–main memory architecture. The evaluation assumes a many-core CPU with non-coherent L1 caches. Compared to an HBM DRAM-based CPU, a monolithic system gets 5.3× and 1.7× higher performance and 6.0× and 1.7× lower memory system energy for several graph and streaming kernels, respectively. We also show that the area savings of co-design permits another 23% in cores and main memory, and that streamlining the LLC/main memory interface incurs a small 4% performance penalty. Last, we also demonstrate that our results will not change appreciably for a many-core CPU with coherent caches.

The rest of this article is organized as follows. Section 2 provides background on ReRAM and describes its monolithic integration with a many-core CPU. Then, Section 3 presents the integrated ReRAM and LLC/directory, and Section 4 presents our shared interconnect and unified controller. After conducting an evaluation in Section 5, Section 6 discusses related work and Section 7 concludes the article.

2 MONOLITHIC CPU–MAIN MEMORY

2.1 Crosspoint Memory Technology

Certain non-volatile memories, such as Intel 3D XPoint [22] and Crossbar 3D ReRAM [11], employ resistive memory cells [10] that can be fabricated without per-cell access transistors. This article considers Crossbar's ReRAM. Crossbar's ReRAM bitcells are based on the creation of metallic filaments in a silicon switching medium [11, 26, 27]. As shown in Figure 2, the ReRAM cells lie in between perpendicular wires, in a *crosspoint architecture* that is fabricated up in the metal stack during **back-end of line (BEOL)** processing steps. Multiple layers of such cells can be fabricated

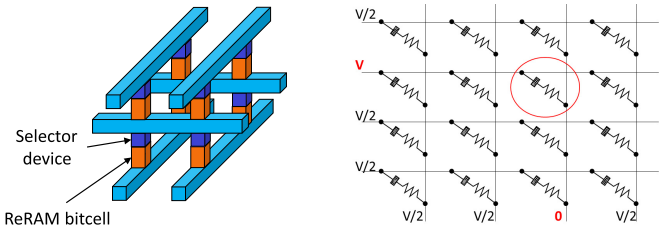

Fig. 2. Closeup of 3D ReRAM bitcells/selectors and crosspoint subarray bias scheme.

to increase density. Figure 2 shows two bitcell layers in between three metal layers. The number of metal layers required to implement n layers of ReRAM is $n + 1$. As many as eight bitcell layers are possible today.

Crosspoint subarrays. Instead of per-cell access transistors, isolation within crosspoint subarrays is achieved via a FAST **Superlinear Threshold Layer (STL)**, enabling high selectivity ($>10^6$–10^{10}). This selector device is integrated in series with the resistive element. A voltage above a threshold ($> V_{TH}$) must be applied across the selector device and ReRAM bitcell to select the cell and perform a read or write.

Figure 2 shows the subarray biasing scheme for selection. All wordlines and bitlines of the subarray are held at V/2, except the selected cell's wordline and bitline are biased to have a difference of V across it. The high selectivity of the selector device ensures minimal sneak current on unselected cells, permitting large subarrays (e.g., 2K × 2K cells). But wordline drive current limitations bound the number of bits that can be sensed along the same wordline to a small number, such as 4 to 8 bits per ReRAM layer. Although cells from different layers can be accessed simultaneously to boost the access parallelism, the biasing scheme in Figure 2 prevents adjacent layers from being accessed. (Non-overlapping crosspoints would have to be accessed to utilize different wordlines and bitlines across adjacent layers; however, unwanted selection of off-crosspoint cells would occur in that case.) So, only every other layer can be accessed. The bottom line is that ReRAM crosspoint subarrays inherently exhibit a fine access granularity.

Crosspoint subarrays are extremely dense due to their lack of access transistors and their fabrication in 3D layers. Based on Crossbar's experience with their own ReRAM memory chips, 64 GB of ReRAM could be fabricated using two-stack subarrays and up to 256 GB of ReRAM could be fabricated using eight-stack subarrays, assuming 14 nm on a 400-mm^2 chip [12]. At the same time, crosspoint subarrays are also quite small. We expect that 64,000 subarrays could fit in a 400-mm^2 die at 14 nm. Each subarray can be accessed independently, providing a very large amount of memory access parallelism.

CPU integration. The use of selector devices instead of per-cell access transistors means that the space beneath individual crosspoint subarrays is *empty*. Access circuitry (i.e., decoders and sense amplifiers) do take up logic circuits, but these are amortized given larger subarrays. For example, at 14 nm and assuming a subarray size of 2K × 2K bitcells per layer and eight layers, we estimate that only 26% of the area underneath each crosspoint subarray would be occupied by the access circuitry, leaving 74% of the area free. Our work exploits this free area to implement the LLC.

Access latency. ReRAM is a relatively new memory technology, so there is quite a bit of uncertainty as to its access latency. This is reflected by the literature where a wide range of latencies have been reported. Read latencies on par with DRAM [47] or even lower than DRAM (below 10 ns) [4, 15] can be found. However, other works have put ReRAM's read latency above that of DRAM, at 120 ns [49], or even between 300 and 600 ns [16]. Although there is agreement that writes are more expensive than reads, a similarly wide range of write latencies have been reported

as well. Write latencies from 10 ns [4] and 35 ns [47] to 150 ns [49] and 203 ns [15] or more [16] can be found in the literature.

In our work, we assume a baseline read latency of 200 ns and a write latency of 400 ns. We believe that this will be achievable when fabricating ReRAM on current state-of-the-art CMOS technology. Our assumptions are based on Crossbar's experience with their own ReRAM memory chips [12]. We do not feel access latencies on par with DRAM will be feasible, especially for the larger subarrays that are required to enable high densities and efficient integration with CPU logic. (Larger subarrays will exhibit greater access latencies due to their long wordlines and bitlines.) Given the uncertainty, however, our experiments will consider varying the access latency.

Write endurance. Besides access latency, another consideration for non-volatile memories is limited endurance. Because monolithic integration enables a highly parallel CPU with a high bandwidth memory system, we can expect elevated write frequencies that make write endurance a particularly important issue. For our evaluation in Section 5, a write endurance of 10^9 write cycles along with good wear leveling [39] would achieve a system lifetime of 8 years. An endurance of 10^{12} to 10^{15} would allow a similar lifetime but without having to use wear leveling. For ReRAM, endurance of 10^6 [49] to 10^{12} [30] have been reported, suggesting that current ReRAM could be used as monolithic main memory but would need wear leveling. Our work does not consider endurance techniques. Instead, we assume wear leveling is employed (e.g., [39]) to permit acceptable lifetimes.

Thermal dissipation. The silicon-based switching material used in Crossbar's ReRAM is very stable across a wide temperature range [11], so we do not expect the on-die ReRAM to impose additional heat dissipation constraints. Additionally, in our technique, the ReRAM is not placed above cores but the LLC, which is generally the coolest part of the CPU. Finally, compared to 3D die stacking, we do not expect heat dissipation to be as problematic for monolithic integration since we only add metal layers on top of a single CPU die. (More heat is trapped between multiple dies in a die stack.) Most of the heat will be produced by the logic in the CPU die's substrate, which is adjacent to a heat sink.

2.2 Compute and Main Memory Architecture

Because of the higher access latency, general-purpose multicore CPUs with a few ILP cores are unlikely to benefit from on-die ReRAM. Instead, we integrate ReRAM into a many-core CPU with a large number of in-order multi-threaded cores that can gainfully use the memory-level parallelism of on-die memory systems. We estimate that 256 cores with four hardware threads each could fit on a die with 256 GB of ReRAM (see Section 5.1). Our cores resemble those found in early **chip multi-threaded (CMT)** achitectures [19, 44], except we add SIMD execution units. The SIMD units not only execute wide memory operations that fetch contiguous data blocks from memory but also execute scatter-gather operations from recent ISAs (e.g., AVX-512 [21]). Such scatter-gather operations are efficiently supported by the fine access granularity of ReRAM. To further increase memory-level parallelism, we also assume that stride prefetching is supported.

Our many-core CPU employs a two-level cache hierarchy with private L1s and a physically distributed, but logically shared, L2. Most CMT architectures with shared L2s support directory-based coherence across the private L1s, but early architectures that considered a large number of cores proposed non-coherent cache hierarchies [29]. (GPUs also employ non-coherent private caches.) We design both a non-coherent and a coherent cache hierarchy. In our evaluation, we simulate the non-coherent hierarchy but show that the results would not change appreciably with cache coherence.

3D SRAM/ReRAM structure. A major benefit of monolithically integrated main memory is physical proximity between the cores and their memory system. To maximize this benefit, we distribute

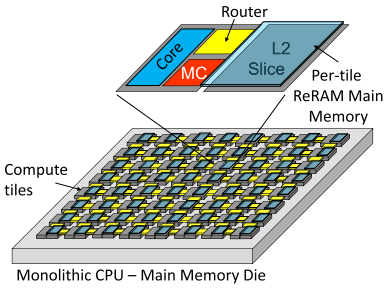

Fig. 3. Distributing ReRAM main memory across tiles of a many-core CPU.

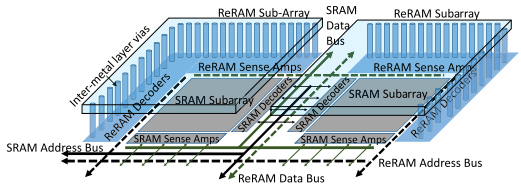

Fig. 4. Co-designed 3D SRAM/ReRAM structure.

the on-die ReRAM across the cores, as shown in Figure 3. We assume a 2D tiled organization, where each tile consists of a core, a private L1 cache (not shown in the figure), and a shared L2 slice. All tiles are interconnected via a 2D mesh **network-on-chip (NOC)**. We added the ReRAM main memory to each tile, allowing a core to access a local portion of main memory that is just 1 or 2 mm away. This virtually eliminates data movement for local accesses. (A core may still access main memory on a remote tile by communicating across the NOC.) In addition, due to the high wire density available on-die, each core can have its own wide connection to the local portion of main memory, resulting in high memory bandwidth aggregated across all cores.

ReRAM subarrays lack access transistors, so they can be integrated over CPU circuits to save area, allowing more cores and main memory to be integrated together. However, not every CPU structure is a good candidate to place beneath the ReRAM. Each ReRAM subarray also has access circuitry that uses the lower metal layers and has dense connections to the upper layers, making routing through it difficult. The need to limit routing constrains the type of circuits that can be placed beneath the ReRAM subarrays.

Previous work has observed that CPU cache, such as the LLC, is a good candidate for 3D integration with the ReRAM main memory [23, 24]. SRAM cache is very regular: the highly interconnected SRAM cells can fit neatly underneath crosspoint arrays and avoid their access circuits, whereas only a few connections (the data and address lines) need to route out, consuming relatively few routing tracks between the ReRAM subarrays. This applies to all levels of cache, but integrating over the LLC provides unique opportunities for further integration discussed in Section 4. We design a 3D SRAM/ReRAM structure, shown in Figure 4, and replicate it to implement an integrated L2 slice and main memory module, shown in Figure 3. The ReRAM cells in this 3D structure exist entirely in the BEOL metal layers, whereas their access circuits are situated at the periphery of the structure. In between the ReRAM access circuits, SRAM subarrays are fabricated with minimal spacing to permit data and address line routing.

Memory controllers. In a conventional CPU or GPU, the memory system is off chip, and access to it is provided via a small number of MCs, much fewer than the number of cores. So, traffic to off-chip memory constricts as it funnels through the MCs, potentially leading to contention. In our monolithic architecture, since the ReRAM main memory is distributed to each tile, it is natural to have one MC per tile, as shown in Figure 3. This eliminates the constriction of memory traffic and provides a highly parallel connection to main memory. Moreover, *having an MC local to each tile also affords opportunities to streamline the LLC/main memory interface.* Our work presents several optimizations related to this interface.

Multi-grain support. Individual ReRAM subarrays only provide a few bits per access. Multiple subarrays must be grouped into ReRAM banks to provide a more useful data access granularity.

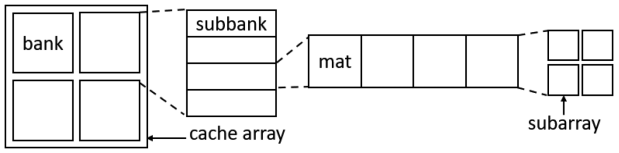

Fig. 5. Sub-dividing an UCA array.

Our design supports the finest access granularity of practical interest, which we assume to be 8 bytes. Assuming eight-stack subarrays, access to four ReRAM layers in parallel, and 4 to 8 bits per accessed layer (see Section 2.1 regarding crosspoint subarray accesses), each subarray can provide 2 to 4 bytes of data per access. This means that we need two to four subarrays to implement each ReRAM bank.

Supporting 8-byte accesses benefits applications with sparse memory access patterns. That said, it is also important to support larger cache block transfers from ReRAM. This can be achieved by activating multiple ReRAM banks as a group. Assuming 64-byte cache blocks, a cache-line fill would require accessing eight ReRAM banks. Multiple schemes for this type of activation have been proposed [48]. In this work, we have opted to break coarse-grain requests into multiple fine-grain requests and collate them within the MSHR file structure (as opposed to sending a single coarse-grain request that would need to wait until *all* eight ReRAM banks are available).

To drive our multi-grain memory system, we employ the following heuristic: cache misses from wide vector load/stores request data at cache-block granularity, whereas cache misses from scatter-gather operations assume low spatial reuse and request data at 8-byte granularity. This determination is made at the L1 cache level based on the cache-missing instruction's opcode and is indicated with a single bit in the access packet header. To handle the multiple request sizes from the memory system, the caches need to be able to store both coarse-grain and fine-grain data blocks. Both the L1 and L2 caches are sectored, and divide their 64-byte cache blocks (i.e., sectors) into eight sub-blocks, each holding 8 bytes.

3 CO-DESIGN STUDY

3.1 Approach

We use Cacti to integrate one **uniform cache access (UCA)** array, or L2 slice, with one local ReRAM main memory module, which we then tile across the many-core CPU. Cacti sub-divides caches for performance [45], as shown in Figure 5. An UCA array is divided into banks that are accessible in parallel. A bank is composed of subbanks; subbanks divide the cache bank bitlines, and only one per bank is accessed at a time. Subbanks are partitioned into mats that divide the wordlines; when a subbank is accessed, every mat is also accessed and provides a fraction of the output. Mats are composed of cache subarrays. They are relatively self-contained with all inputs and outputs coming through their center. The cache mat is the unit of SRAM that we integrate underneath ReRAM banks.

For an associative cache, the number of sets per wordline can be adjusted: if it is a whole number, multiple sets will be mapped to the same wordline; if it is a fraction, the ways of one set will be mapped to different consecutive wordlines. This parameter, s, allows adjusting the aspect ratio of the mats as seen in Figure 6(a). We use mat size and the s parameter to design SRAMs that will fit beneath the ReRAM banks.

To implement our multi-grain memory system, we use ReRAM banks comprised of two to four ReRAM subarrays each. Assuming a per-bank access width of 8 bytes, we gang eight ReRAM banks together to yield an access width of 64 bytes. Cache sectoring can be implemented at the subbank

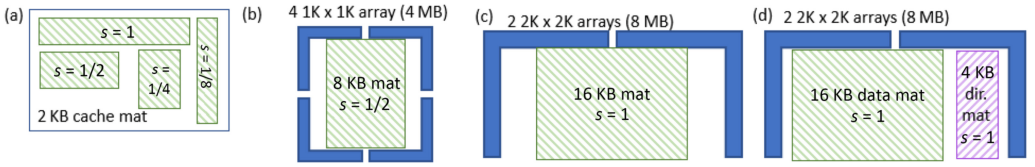

Fig. 6. (a) A sweep through the s parameter that controls the number of sets per line. (b, c) Relative sizing of cache mats and ReRAM subarrays. (d) A data mat and directory mat beneath ReRAM subarrays.

level by dividing each subbank into eight mats, one per sub-block within a sector, with each mat supplying 8 bytes of data. Then, we integrate the eight cache mats underneath the eight ReRAM banks in an "output matched" fashion such that the 8-byte ReRAM bank access width aligns with the 8 bytes stored in each cache mat. (This eliminates any planar data movement when transferring data between the ReRAM banks and SRAM.) For fine-grained transactions, we activate just one cache mat/ReRAM bank; for coarse-grained transactions, we activate all eight cache mats/ReRAM banks. Our co-design study uses Cacti's smallest supported feature size of 32 nm.

3.2 Cache Mat and ReRAM Bank Size

First, we used Cacti to find mat sizes that would best fit under ReRAM banks. We assumed that each ReRAM subarray's access circuits form an "L-shape" along two edges of the subarray; the remainder of the subarray area is available for integrating the cache mat. In Cacti, the cache was configured to have one cache bank, one subbank, and eight mats with eight-way set associativity and 64-byte cache blocks. The cache capacity and sets per wordline were swept to find mats that could be placed beneath ReRAM banks. We focused on 2K ×2K and 1K ×1K ReRAM subarrays in groups of two or four per ReRAM bank to achieve an output size of 8 bytes. The overhead of the access circuitry increases in smaller ReRAM subarrays, leaving insufficient area to reasonably integrate the cache mats. For larger ReRAM subarrays, the delay and energy required per access increases steeply, making them poor candidates for main memory. So, we limit our study to 2K × 2K and 1K × 1K ReRAM subarrays.

Figure 6(b) and (c) show example cache mats and ReRAM subarrays drawn to scale for the four- and two-subarray ReRAM banks, respectively. The drawings look down "through" the ReRAM bitcell stacks, with the blue "L shapes" denoting each ReRAM subarray's access circuits. (We note that Figure 4 is a 3D rendering of the top half of Figure 6(b).) Because we limited both the cache and ReRAM subarrays to powers of 2, there are some unused areas, as Figure 6(b) and (c) show.

Next, Cacti was modified to include the area overhead of the ReRAM subarrays. This includes not only the ReRAM's access circuits but also the interconnect for routing their address and data lines. The additional area overhead expands the placement of the cache mats and their interconnects, thus impacting the cache energy and access time. (These modifications do not attempt to model the energy used by the ReRAM, but rather the impact the ReRAM has on the cache.) Figure 7(a) shows the area of a cache-only design for an eight-mat 128-KB subbank using the cache mat from Figure 6(c). Figure 7(b) shows the co-design of the same 128-KB cache subbank with eight ReRAM banks in an output matched fashion. All eight ReRAM banks in Figure 7(b) are accessed for coarse-grained (64-byte) transactions, but each ReRAM bank can be accessed independently for fine-grained (8-byte) transactions.

3.3 L2 Slice and Main Memory Module

After co-designing the cache mat and ReRAM bank pairs, the cache capacity was increased by replicating the cache mat/ReRAM bank units from Figure 6(b) and (c). The number of total subbanks

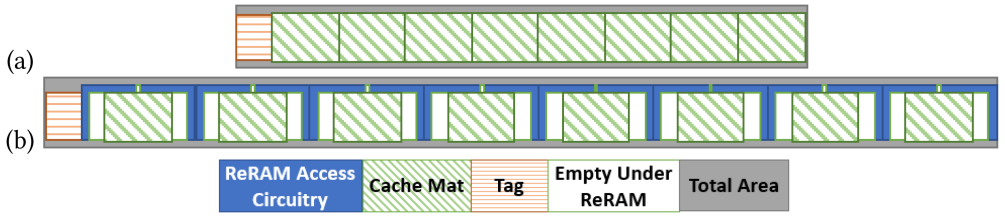

Fig. 7. (a) A 128-KB cache subbank with eight mats. (b) The same co-designed with eight ReRAM banks.

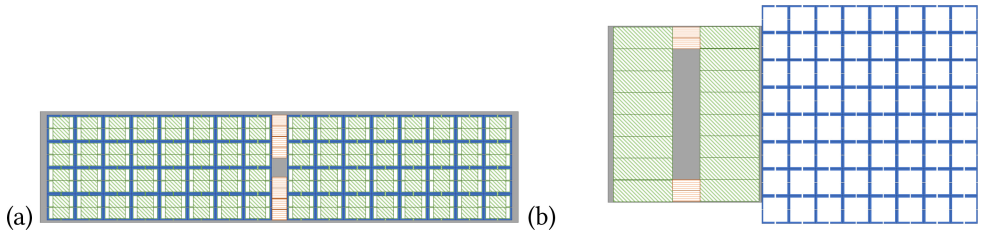

Fig. 8. (a) A 2-MB L2 slice organized as two cache banks in a 1 × 2 grid when co-designed with 128 8-MB ReRAM banks. (b) The same amount of cache and main memory designed separately. (Drawn to scale.)

for each design was increased to 32 and 16, respectively, with eight mats per subbank while maintaining the mat dimensions. This yielded 256 4-MB ReRAM banks and 128 8-MB ReRAM banks per L2 slice, respectively, which provides a 1-GB main memory module over a 2-MB L2 slice.

We further modified Cacti to allow us to dictate the layout of banks. Cacti's default placement of banks in a square is based on the assumption that the banks will also be close to square; our co-designed banks often break this assumption. Using our modified Cacti, we find the optimal configuration for an L2 slice-main memory module. Mat dimensions were fixed based on their co-design with a ReRAM bank, the mats/subbank were fixed to provide the proper data access granularity, and the total number of subbanks was fixed to maintain the capacity and approximate area. The number of cache banks and their layout (e.g., 4 ×4 banks or 2 ×8 banks) were optimized within these constraints.

Figure 9 shows the energy and percent of the cache occupied by ReRAM subarrays (which we call *ReRAM area efficiency*) for the layouts considered; in this analysis, delay is correlated with dynamic energy ($r^2 = 0.999$). A few of the design points in the figure are labeled with their layout dimensions, showing that bank layouts with aspect ratios closest to squares produced the best energy efficiency. A number of other factors, including the size of the tag arrays and the direction of the bulk of the routing (vertical or horizontal), affect the ReRAM area efficiency. Figure 8(a) shows Cacti's best layout for the 2-MB L2 slice. This design consists of two cache banks with eight subbanks each, in a 1 × 2 layout, integrated with 128 8-MB ReRAM banks.

3.4 Co-Design Impact

The main benefit of co-design is area savings. For example, Figure 8(b) shows the area of the 2-MB L2 slice when designed separately (not integrated with) the 128-bank ReRAM, which has an area of 10.6 mm^2 at 32 nm. Comparing Figure 8(a) and (b), we see that co-design reduces the total area to 8.3 mm^2, a 27% area reduction compared to a separate design.

However, co-design negatively impacts the L2 slice due to two key changes: the modification of the dimensions of the mats to fit beneath the ReRAM banks, and the area overhead of the

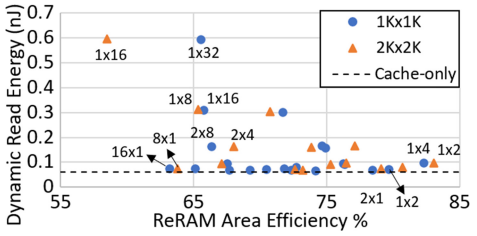

Fig. 9. Design space for a 2-MB L2 slice with co-designed ReRAM. The line represents a cache-only design's dynamic read energy.

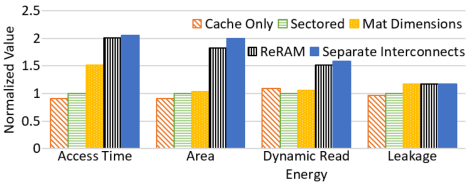

Fig. 10. Penalty for co-designed L2 slice normalized to a cache-only design.

ReRAM banks and their internal interconnects. Figure 10 shows the overheads associated with each of these factors for the L2 slice in Figure 8(a), as well as the overheads of the sectored cache.

The sectored cache requires adding additional valid and dirty bits to the tag array. This has a minimal impact of the cache, increasing the area by 10%, the access time by 9%, and the leakage by 3%. Cacti finds it optimal to divide the tag array into small mats due to the increased capacity, which does result in a dynamic energy savings of 9%.

The change from the sectored cache-only design to the co-design mat size primarily increased the cache access time. The access time remains less than 4 ns, and the latency of a distributed, shared cache is dominated by the time spent reaching the cache rather than the access time once there [37], minimizing its impact. Increasing the number of divisions also increased leakage by an additional 18% since control logic and number of drivers increased.

Adding the area overhead of the ReRAM had the biggest impact, increasing the cache area by a total of 116% over the standard cache. This would only be a concern if the total design area increased, but co-design results in a significant net savings when considering the area of the L2 *and* main memory. Finally, dynamic access energy increases by a total of 45%. Part of this increase is due to having a separate interconnect for ReRAM accesses (which adds 7% more access energy compared to the cache-only design). This can be mitigated by modifying the LLC/main memory interface, discussed in the next section. More importantly, Section 5 will show that the dynamic energy impacts to the LLC are negligible when put in the context of total memory system power, making the area savings of co-design well worth it.

3.5 Cache Coherence Directory

Although the architecture we evaluate in Section 5 does not include coherence, the co-designed cache-ReRAM could support it with minimal changes. It is possible to implement a cache coherence directory along with each L2 slice underneath the ReRAM. Rather than re-design from scratch, we start with the best layout for the integrated L2 slice–main memory module in Figure 8(a), keeping all parameters from Cacti's solution fixed. Due to the power-of-2 and aspect ratio constraints, the area underneath the ReRAM is not entirely consumed by the L2 slice SRAM. So, we can design another set of smaller mats for the coherence directory, utilizing the remaining free area.

Because our L2 is a shared cache, the directory would be for L1 coherence. Given the smaller size of the L1 caches, for simplicity, we assume a sparse directory [18] with full-map directory entries. Each directory entry contains a 32-byte bit vector that can track up to 256 sharers. Similar to the L2 slice, we created an eight-mat subbank for the directory, and integrated it underneath

eight ReRAM banks. This means that each directory mat has an access width of 4 bytes. We further assume the directory is 4-way set associative.

We used Cacti to find the largest directory mat with the desired access width and associativity that will fit in the spare area under each ReRAM bank. Cacti selected a 4-KB directory mat with $s = 1$. Figure 6(d) illustrates the layout of this directory mat next to the cache mat. For the entire L2 slice–main memory module in Figure 8(a), Cacti is able to create a 512-KB directory. (Although not shown, the tags for this directory also fit next to the L2 tags shown in Figure 8(a).) This is large enough to handle a 32-KB, 64-KB, or 128-KB L1 with 64×, 32×, or 16× over-provisioning, respectively.

Although the directory mats fit underneath the ReRAM, there is still an impact for the directory's interconnection with the cache controller (albeit, with a narrower 32-byte data bus compared to the 64-byte data bus needed for the L2 and main memory). Figure 4 shows the two separate interconnects for the L2 slice and main memory. If a third interconnect were added for the directory, the area of the L2 slice–main memory module in Figure 8(a) would expand by 5.1%. This would also slightly impact the L2 overheads reported in Figure 10.

4 LLC/MAIN MEMORY INTERFACE

In addition to co-designing the cache and directory mats with ReRAM subarrays, monolithic CPU–main memory integration also provides an opportunity to streamline the interface between the LLC and main memory. Streamlining the LLC/main memory interface can provide further area savings as well as energy benefits. There are two major system components that we consider reducing: interconnects and controllers.

4.1 Shared Interconnects

If main memory moves onto the CPU die, there is an opportunity to eliminate interconnects that facilitated communication between what used to be discrete components. Of course, the system memory bus can be eliminated because memory requests no longer need to cross the system motherboard.

But in addition, there are also the internal interconnects that facilitate communication from controllers to the cache and directory mats/ReRAM banks. Normally, the LLC, directory, and main memory are discrete components, so there are distinct internal interconnects for each of them. But most of the time, the main memory's internal interconnect (for the ReRAM banks) is underutilized. In the design we have laid out, there are 128 ReRAM banks and they have read latencies of 200 ns, meaning that even when fully accessing all of the ReRAM banks, only 128 requests would be sent in 200 ns, a time frame that could service 200 requests.

For the co-designed structure from Section 3, the cache and directory access traffic and ReRAM access traffic go to the same place. So, the cache controller and ReRAM MC could share internal interconnects. In particular, the non-coherent L2 slice–main memory module shown in Figure 4 and designed in Section 3.3 could go from two interconnects down to one interconnect. In addition, the L2 slice–main memory module with a directory designed in Section 3.5 could go from three interconnects down to two interconnects.

Sharing interconnects results in modest savings in access time and energy. The main benefit, however, is area savings. Without a directory, the area of the L2 slice–main memory module from Figure 8(a) reduces by 12%. When combined with the 27% area savings from co-design mentioned in Section 3.4, there is a total area savings of 39%. With a directory, the area of the L2 slice–main memory module reduces by 8.8%. But sharing interconnects can hurt performance by introducing contention. We will evaluate this in Section 5.

Table 1. McPAT and Cacti Area at 14 nm

Component	Area (mm^2)
Core (including L1 cache)	0.72
Router	0.13
Memory controller	0.30
L2 slice-ReRAM module (co-designed)	1.46
L2 slice	0.81
ReRAM module	1.25

4.2 Unified Controller

Another streamlining opportunity is merging the cache controller and MC. The cache controller schedules accesses to the cache arrays and keeps track of outstanding misses, whereas the MC schedules the accesses to the memory arrays for those cache misses. The two can be combined by making the cache's MSHRs and the MC's scheduling queue (which provides the controller's functionalities) the same structure. Instead of forwarding the request to the main MC, the unified controller sends them directly to the main memory bank. This eliminates communication between the cache controller and MC, reducing total access delay. Contention at the cache controller is minimally affected by combining the two controllers, as every request would need to be handled by the cache controller regardless of unification.

Merging both the interconnect and the controllers allows for even less data movement. In the case of a cache miss, the request can be passed from the cache array to the memory array without going across the internal interconnect. Similarly, when the data is available from the memory, it can immediately be written into the cache array. This saves a traversal of the internal interconnect for every cache miss, and two traversals in the case where it can be serviced immediately by the memory. This requires the miss handling portion of the cache pipeline to tightly incorporate memory operations. When using a unified interconnect, the pipeline scheduling must accommodate the resource usage when sending and receiving data to/from the ReRAM banks. Additionally, the miss handling portion of the pipeline must schedule the memory requests that cannot be serviced immediately.

5 EXPERIMENTAL EVALUATION

This section evaluates our monolithically integrated CPU–main memory chip. The evaluation assumes that our CPU employs the co-designed L2–ReRAM main memory module from Sections 3.1 through 3.4 without the directory, but with the shared internal interconnect and unified controller from Section 4. At the end of the evaluation, we will discuss the impact that including cache coherence would have on our results.

5.1 Chip Area Simulated

Our evaluation assumes 14 nm. We first compute the CPU size that this technology node enables. From Section 3, we have Cacti's area estimates for our co-designed L2 slice–ReRAM main memory module. For the core, we used McPAT's Niagra 2 model [33], which is an in-order multi-threaded core. The McPAT model not only includes the core (with L1 I-/D-cache), but it also includes the router and MC. The MC is for DRAM; we removed the area for the I/Os and used the reduced model as an estimate for a unified cache-ReRAM controller. We performed a regression on the technology nodes available in Cacti and McPAT to scale down the component areas to 14 nm. The first four lines in Table 1 report these results. We also did a regression on the cache-only design in Figure 8(b). The last two lines in Table 1 report these results.

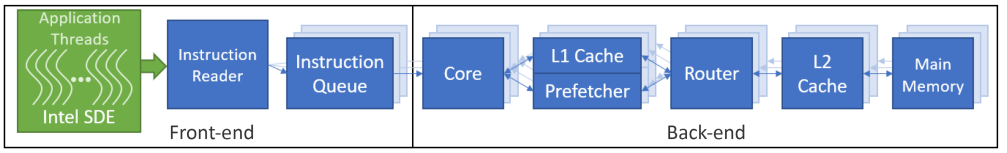

Fig. 11. Overview of our simulator.

Table 2. Simulation Parameters for the Experiments

Cores	256, 4-way multi-threaded
Clock Rate	2 GHz
L1 Cache	32 KB, 4-way, 1 cycle
L2 Cache Slice	256 KB, 8-way
Sector Size	64 bytes with 8-byte sub-blocks
Stride Prefetcher	32-entry stride table
Unified Controllers	256 (1 per core), 64 MSHRs
ReRAM Banks	32,768 (128 per tile)
ReRAM Read/WriteLatency	200/400 ns
ReRAM Access Granularity	8 bytes
On-Chip Network	16 × 16, 2D-Mesh
Network Channels	4 × 64 bytes

Table 1 suggests that our co-designed L2 slice–ReRAM module permits a many-core CPU with 256 cores/tiles to fit in a 25.8 × 25.8 mm die, similar to that of Intel's Knights Landing [13]. This would provide a 256-GB main memory system. Table 1 also shows that a separately designed cache and main memory would result in a tile that is 23.0% larger. Section 5.6.2 will explore this issue.

5.2 Simulator

We created a simulator that models the many-core CPU from Section 5.1 with an on-die main memory system. A high-level block diagram of our simulator is presented in Figure 11. The simulator faithfully models our co-designed L2–ReRAM main memory module and streamlined LLC/main memory interface. Table 2 lists the simulation parameters used as the baseline for our performance evaluation.

Similar to recent simulators of many-core CPUs with a very large number of threads [7, 36, 43], we use an Intel Pin-based front-end [35] to feed a parallel instruction trace to a cycle-accurate back-end. The main difference is that for the front-end, we employ Intel's **Software Development Emulator (SDE)** [20]. SDE can execute ×86 binaries with thousands of threads. It can also emulate AVX-512 instructions that support scatter-gather operations, which we assume (see Section 2.2). Unlike previous multi-core simulators, ours sacrifices simulation speed in favor of cycle accuracy with respect to the memory system; depending on the memory intensity and performance, the simulator simulates at a rate of 5 to 20 KIPS.

Cores. The front-end executes 1,024 threads in SDE and maps them in groups of four onto the 256 back-end cores. Cores are single issue, in order, and multi-threaded with a five-cycle context switching overhead. The cores employ a simple one cycle per instruction model and stall threads whenever they encounter a cache miss. When encountering a scatter-gather instruction, the core will perform each access on consecutive cycles and the thread will only stall once all requests have been sent. In the literature, it has been shown that one cycle per instruction simulation models

work well for relative performance when focused on homogeneous cores working on homogeneous workloads [6]. All of our simulations meet these requirements.

L1 cache. The L1 cache is 32-KB/tile and 4-way associative. Hardware prefetching is performed via a 32-entry 2-delta stride table. After stride detection, data prefetching occurs into 16 prefetch buffers. The L1 cache is explicitly flushed when the core reaches a barrier to enforce coherence across barriers. When a cache miss occurs, the L1 cache uses the address to calculate the coordinates of the L2 slice within the 2D mesh network where the data is stored and sends a request. If the network buffer is full, it will stall the core and attempt to send it again during the next cycle. All L1 caches are sectored to allow cache-miss fills at either 8-byte (on scatter-gather misses) or 64-byte (on all other misses) granularity.

Network-on-chip. The tiles are interconnected via a 16×16 2D mesh network. The NoC carries the traffic between the private L1s and the L2 slices on L1 cache misses. It employs dimension-ordered routing and supports two channels (one for requests and one for replies), and is deadlock free. The routing and request information is contained in an 8-byte header sent with every packet. We assume that each channel between any pair of NOC routers contains two uni-directional 64-byte sub-channels, resulting in a bisection of 32,768 wires. Although this is quite wide, it is feasible given that all routers are integrated on the same die.

L2 cache/ReRAM. The L2 cache is 256-KB/tile and 8-way associative. The simulated L2 slices are smaller than the slices designed in Section 3.4 to account for the much smaller data inputs we are able to simulate. Like the L1, the L2 cache is sectored to allow cache-miss fills at either 8-byte or 64-byte granularity, as determined by a bit in the request header. Each cache slice has 32 MSHRs. Each slice also contains 128 ReRAM banks, for a total of 32,768 banks chip-wide. ReRAM reads take 200 ns, whereas writes take 400 ns. The simulator faithfully models our co-designed LLC/ReRAM main memory module and streamlined LLC/main memory interface, including all queuing and conflicts at the controller.

Cache and memory scheduling follows a first-in first-out approach with older requests having priority (practically meaning main memory requests have precedence over cache requests, and receiving data from arrays has precedence over sending new requests). Cache read hits take five cycles, and writes take seven cycles. The initial cycle for each consumes the shared interconnect, preventing any ReRAM requests from being sent to the banks. The initial handling of a miss requires three cycles, eventually followed by a seven-cycle write when the data is received. If no MSHRs are available, handling new requests is stalled until main memory requests are completed and an MSHR is freed. Scheduling ReRAM requests takes place during a miss while accessing the MSHRs and is performed in a first-in first-out manner for each ReRAM bank. As each ReRAM request completes, the next oldest request to the now unoccupied bank is sent, consuming a cycle on the shared interconnect. The cache operations are pipelined; sending data to the arrays, tag lookup, data array access, and MSHR operations are each assigned a pipeline stage to allow multiple requests to be serviced simultaneously.

DRAM. We compare a monolithically integrated CPU–main memory chip to a discrete system composed of the same CPU connected to four HBM2 DRAM stacks with no ReRAM. When a discrete DRAM system is being modeled, an additional two network channels are added from the L2 cache to the MCs, and the main memory portion of the simulator is replaced with DRAMSim3 [34], which accurately models all DRAM components. HBM2 uses four or eight MCs per stack—located in the middle of each edge of the NOC—that control 16 or 32 total DRAM channels. The four-MC case employs a 128-byte cache-block size and always fetches cache blocks. The eight-MC case employs a 64-byte cache-block size and normally fetches cache blocks, but uses a 32-byte fetch size (the finest granularity in HBM2) for scatter-gather accesses. The peak DRAM bandwidth in both cases is 1 TB/s. We also consider a scaled-up system with eight HBM2 stacks on 32 or 64 channels,

Table 3. Benchmark Names, Input Sizes, and Number of Instructions Simulated (in Billions)

Graph Kernels		
All Pairs Shortest Path (APSP)	scale = 22	1.4
Betweenness Centrality (BC)	scale = 22	1.5
Page Rank (PR)	scale = 22	2.1
Single Source Shortest Path (SSSP)	scale = 22	1.8
Streaming Computations		
DAXPY	1.47B elements	1.5
K-means (KM)	1M points, 32 features	1.0
Nearest Neighbor (NN)	204.8M hurricanes	1.3
Pathfinder (PF)	15M columns, 100 rows	2.4

and a peak bandwidth of 2 TB/s. To account for the difference in area between our co-designed cache and off-chip DRAM, each tile has a 512-KB LLC slice, double that of the monolithically integrated cache.

5.3 Benchmarks

We drive our simulations using eight benchmarks, which are listed in Table 3. Half of our benchmarks are graph kernels, whereas the other half are streaming computations. All but one are from standard benchmark suites—either CRONO [2] or Rodinia [9]. The exception is DAXPY, which computes $y[i] = a * x[i] + y[i]$ and was written by the authors.

All of the kernels were explicitly parallelized to create threaded code. The threaded code was then vectorized by hand using AVX-512 intrinsics to generate SIMD instructions. Unit-stride array traversals occur in all of the benchmarks and were converted to packed vector load/store instructions. In the graph kernels, memory indirection through edge lists are ubiquitous and were converted to scatter-gather memory instructions. There is also opportunity for scatter-gather in one of the streaming benchmarks, NN. In the streaming computations, we were able to partition the arrays that exhibited unit-stride traversals such that the array portion operated upon by each core is allocated in the core's local ReRAM memory. However, the graph kernels exhibited large amounts of irregular accesses, accessing data across a large portion of the memory space, preventing them from being partitioned this way.

Because the simulator assumes the non-coherent version of our LLC–ReRAM main memory, we must manage coherence manually. All of our benchmarks execute fully parallel loops, with dependences occurring only across barriers. We added software directives to flush the L1 D-cache right before each thread enters a barrier. Our simulator simulates these software-induced L1 flushes, including writebacks of dirty L1 blocks.

The second column of Table 3 specifies the inputs for each benchmark. For the graph kernels, we created an input graph using SSCA2 [5], which is based on the **Recursive MATrix (R-MAT)** scale-free graph generator [8]. The number of vertices in the generated graph is 2^{22} (i.e., scale = 22). For the Rodinia benchmarks, the input sizes are scaled-up versions of the inputs provided with the benchmarks [9] to match the high degrees of parallelism in our experiments.

We functionally execute the serial initialization code of each benchmark in SDE only, then turn on the cycle-accurate back-end for the parallel region. The last column of Table 3 reports the number of instructions (in billions) simulated in the parallel region. For the streaming benchmarks, these represent the entire parallel region. For the graph kernels, the instruction counts represent

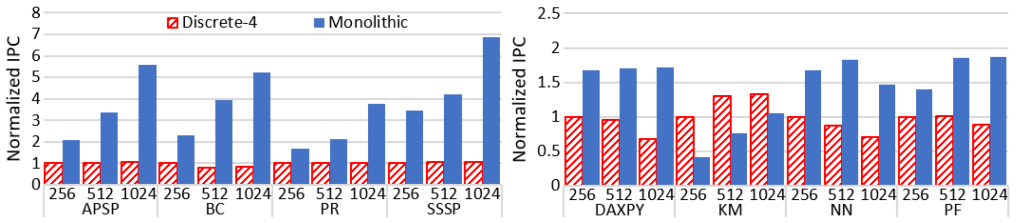

Fig. 12. Normalized throughput on a monolithically integrated CPU–main memory chip and on a discrete system with four-stack HBM2.

a portion of the parallel region. All benchmarks exhibit the same behavior throughout the parallel region, so we expect the four partially simulated benchmarks to still yield representative results.

5.4 Performance Results

Figure 12 presents our performance results. The graph kernels are shown on the left side of the figure, whereas the streaming computations are shown on the right side. The figure plots throughput on the many-core CPU with a monolithically integrated ReRAM memory system and with four discrete HBM2 DRAM stacks, labeled "Monolithic" and "Discrete-4," respectively. The graph kernels use the fine-grained 32-channel HBM2, whereas streaming computations use the coarse-grained 16-channel HBM2, which provide the best performance for each benchmark. We scale the number of threads from 256 to 1,024 by running 1, 2, or 4 threads in each multi-threaded core. All bars are normalized against the Discrete-4 bars at 256 threads.

The monolithic system significantly outperforms the discrete HBM2 DRAM system. At 1,024 threads, the monolithic system's throughput for the graph kernels is 5.3× higher than HBM2 on average, whereas the monolithic system's throughput for the streaming computations is 1.7× higher than HBM2 on average. Different reasons account for the performance advantages in the graph kernels versus the streaming computations.

For graphs, the majority of cache misses are incurred by indirect memory references through scatter-gather operations. The combination of 1,024 threads and 8-way scatter-gather generates an enormous number of memory requests. The monolithic system employs 8-byte fetches for scatter-gather memory requests. This opens up the 32,768-way bank-level parallelism to support the indirect memory references. So, the ReRAM-based memory system is able to keep up without incurring much bank contention. In Figure 12, this enables the performance scaling on the monolithic system from 256 to 1,024 threads for the graph kernels. In contrast, HBM2 is unable to handle the massive parallelism. It only has 256 banks, and although it uses HBM2's finest fetch size (32 bytes) on sparse requests, this still transfers unused data and incurs contention. Therefore, HBM2's performance does not scale.

For streaming, the majority of cache misses are incurred by linear array traversals. Instead of random memory access parallelism, high streaming bandwidth is more important. The monolithic system adapts to a cache-block fetch granularity, using its prefetcher to create long streams of cache-block requests. Moreover, each core primarily computes on data stored in its local ReRAM module (see Section 5.3). So, the streaming accesses avoid the NOC and experience minimal contention, allowing the monolithic system to almost achieve its peak bandwidth, 1.3 TB/s (32,768 banks × 8 bytes / 200 ns) as can be seen in Figure 13. Note that the effectiveness of prefetching is evident in Figure 12: higher thread counts provide less latency tolerance benefits for the streaming computations compared to the graph kernels.

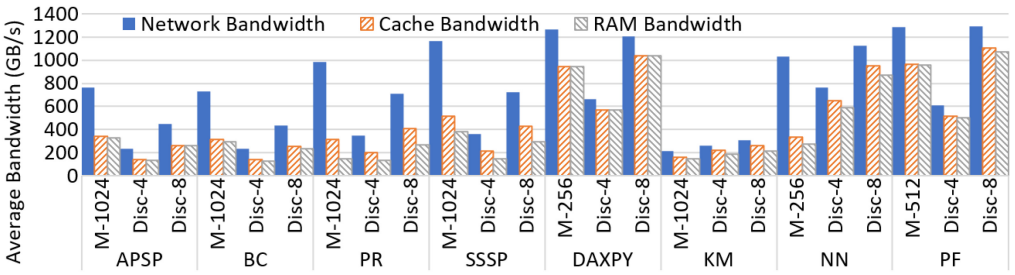

Fig. 13. Average bandwidth in the network, LLCs, and RAM for a monolithically integrated CPU–main memory chip (running 1,024 threads) and for discrete systems with four-stack or eight-stack HBM2.

In contrast, although the HBM2 memory system has a comparable peak bandwidth, 1 TB/s, it requires row buffer locality to achieve the peak. Streaming usually exhibits high spatial reuse, but our many-core CPU causes a large number of streams to interleave at the small number of DRAM banks, destroying the reuse within row buffers. As a result, only a fraction of the peak bandwidth is achieved. In fact, for DAXPY, NN, and PF, performance actually *degrades* with more threads, as this causes even greater stream interleaving.

Two benchmarks, NN and KM, exhibit unusual behavior. Although most streams are dense, NN performs striding. For the monolithic system, the striding pattern permits only 25% of the ReRAM banks to be utilized, reducing peak bandwidth by $\frac{3}{4}$. This limits performance gains for the monolithic system as thread count scales (performance even goes down from 512 to 1,024 threads). Last, the one benchmark for which HBM2 outperforms our monolithic system is KM. KM has a high cache-hit rate and is much less memory intensive. It is latency bound and hence prefers HBM2.

The average bandwidth of the network, cache, and main memory are shown in Figure 13. We only show the thread count for which DRAM performs the best per benchmark. The network bandwidth includes a request header, increasing its bandwidth usage over the cache bandwidth. This is most evident in benchmarks that use scatter-gather requests since each packet will have a larger ratio of header information to data. The cache and RAM bandwidths are very similar for the majority of benchmarks due to high L2 cache miss rates. The exceptions are PR and SSSP, which exhibit a degree of locality and so benefit from the LLC, resulting in their cache bandwidth being larger than their RAM bandwidth. The issue of interleaving streams is evident for the discrete systems, as they only reach about half their peak bandwidths in the best cases for the streaming benchmarks.

Figure 14 shows a comparison against a more aggressive HBM2 with eight die stacks and 32 controllers, labeled "Discrete-8." Instead of showing all thread counts, we only show the thread count for which DRAM performs the best per benchmark. For graphs, HBM2 roughly doubles in performance, but the monolithic system is still 2.7× faster. For streaming, the faster HBM2 now outperforms the monolithic system in all cases except NN. NN is not write intensive, and much of its write traffic is sent to the $\frac{3}{4}$ of banks that are not accessed by reads due to NN's striding pattern. Reducing write contention, especially as writes are twice the latency of reads, gives NN an advantage over the other memory-intensive streaming benchmarks in the monolithic system. Even though HBM2 outperforms the monolithic system in most streaming benchmarks, the monolithic system is still within 9.7% of the aggressive HBM2. This underscores the advantage of on-die memory systems: physical locality and memory-level parallelism allow them to be competitive in performance with an off-chip memory system that has almost twice the peak bandwidth.

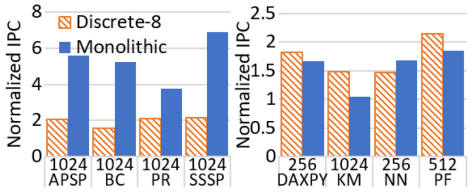

Fig. 14. Comparing normalized throughput with eight-stack HBM2.

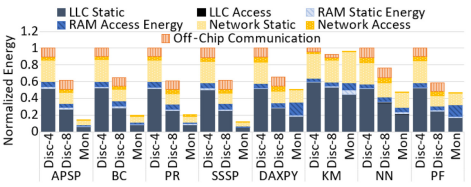

Fig. 15. Dynamic, refresh, and static energy in the memory system.

5.5 Energy Results

Figure 15 shows the energy in the memory system, including the LLC, network, data movement off the CPU chip, and RAM access (DRAM or ReRAM) for the four-stack HBM2, eight-stack HBM2, and monolithic systems, labeled "Disc-4," "Disc-8," and "Mon," respectively. Once again, we only report results for the best-performing thread count per benchmark. Varying thread count only affects the dynamic energy by a few percent, as roughly the same total amount of work is performed; however, static energy grows with execution time. To obtain the LLC's energy, we performed a regression on the Cacti results from 90 nm to 32 nm, then extrapolated the 32-nm design from Section 3.4 to 14 nm. We did this for the co-designed cache with a shared interconnect (monolithic system) and the cache-only design (HBM2 system). For data movement on the CPU chip (i.e., the NOC), we assumed 0.1 pJ/bit/mm. For off-chip data movement to HBM2, we used 2.8 pJ/bit [38]. DRAMSim3 provided the DRAM energy. Finally, similar to latency, there is also uncertainty in ReRAM's access energy. In recent literature, 1pJ/bit [4] and 1.6pJ/bit [49] have been reported. Based on Crossbar's experience with their ReRAM technology, we assume 2.4 and 4.8 pJ/bit for ReRAM's read and write energies [12].

As Figure 15 shows, the monolithic system consumes less energy than HBM2. On average, for the graph kernels, the monolithic system uses 6.0× and 3.7× less energy compared to four- and eight-stack HBM2, respectively, and for the streaming computations, the monolithic system uses 1.7× and 1.2× less energy compared to four- and eight-stack HBM2, respectively. The monolithic system does not incur any off-chip data movement. For the streaming computations, the monolithic system eliminates most of the on-chip data movement as well since memory accesses are almost entirely localized within each tile. Additionally, as discussed in Section 5.4, the graph kernels and NN exhibit sparse memory access patterns for which HBM2 incurs over-fetch. This increases all components of HBM2 energy consumption relative to the monolithic system in these benchmarks.

5.6 Detailed Design Results

5.6.1 Latency Sensitivity. As discussed in Section 2.1, ReRAM is a relatively new memory technology, so there is uncertainty in its access latency. To study the sensitivity of our results to latency, we ran simulations with read latencies of 100, 200, 500, 750, and 1,000 ns. In these experiments, the write latencies are always set 2× higher. Figure 16 reports the monolithic system's throughput with 1,024 threads for the 200-, 500-, and 750-ns experiments as well as the best eight-stack HBM2 throughput normalized to the best four-stack HBM2 throughput (marked by the dashed line). Using all five latency experiments per benchmark, we performed a regression to extrapolate the latency for which the monolithic system achieves the *same throughput* as HBM2. The two values shown above each set of bars report this "break-even latency" for the monolithic system compared to four- and eight-stack HBM2.

In the graph kernels, the monolithic system remains superior to four- and eight-stack HBM2 even at higher latencies. Averaged across the four graph kernels, the monolithic system's

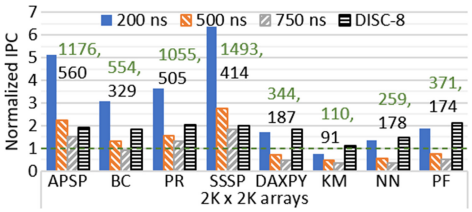

Fig. 16. Latency sensitivity results for 2K × 2K subarrays and 128 banks/tile.

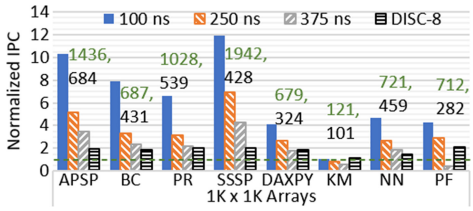

Fig. 17. Latency sensitivity results for 1K × 1K subarrays and 256 banks/tile.

break-even latencies with four- and eight-stack HBM2 are 1069 ns and 504 ns, respectively, showing that there is significant "latency headroom" in the graph kernels. For streaming, the monolithic system beats four-stack HBM2 at 200 ns, but loses at 500 ns or higher. Nevertheless, there is still a little headroom: averaged across the four streaming computations, the monolithic system's break-even latency with four-stack HBM2 is 271 ns. However, to break even with eight-stack HBM2, ReRAM's latency would need to drop below 200 ns: 157 ns, as shown in Figure 16.

An important factor affecting access latency for ReRAM is the subarray size. We expect that the latency for 1K × 1K ReRAM arrays will be a little more than half that of 2K × 2K ReRAM arrays. Using the 1K × 1K ReRAM arrays also results in an increased amount of parallelism, with 256 ReRAM banks/tile. Thus, to study the performance of 1K × 1K ReRAM arrays, we simulated 256 banks/tile at half the latencies simulated for 2K × 2K ReRAM arrays. (In other words, we simulated read latencies of 50, 100, 250, 375, and 500 ns, plotting the middle three results in Figure 17, then using all five results for regression analyses, similar to what we did for the 2K × 2K results.) Figure 17 shows these results. The 1K × 1K configuration has a larger negative impact on the cache. Although the cache area is only 1.2% larger than the 128 bank/tile slice, the smaller mats required to fit beneath the smaller ReRAM banks increase leakage by 28% rather than 18%, as more access circuitry is required. However, in return, Figure 17 shows the throughput increases significantly compared to Figure 16. Moreover, the graph kernel average break-even latencies with four- and eight-stack HBM2 increase to 1,273 ns and 603 ns, respectively. For the streaming computations, the average break-even latencies with four- and eight-stack HBM2 increase to 558 ns and 292 ns, respectively. The higher break-even latencies are due to the increased parallelism from the larger number of ReRAM banks.

We also conducted experiments that vary the write latency while keeping read latency fixed. This can occur when addressing reliability issues: the system may perform multiple cycles of writing and checking data integrity. Figure 18 reports the throughput on a monolithic computer when the write latency is increased from 400 ns to 2 μs with the read latency held at 200 ns. Each benchmark is run with 1,024 threads and the performance is normalized to the baseline configuration with 400-ns writes. These results show that the performance of our benchmarks is much less sensitive to increases in write latency alone.

If the read latency is maintained at 200 ns, the break-even write latency for the graph kernels compared to the four- and eight-stack HBM2 is 8.3 μs and 6.4 μs, respectively. For streaming computations, 1.4 μs is the break-even write latency compared to the four-stack HBM2; changing the write latency alone cannot make the streaming computations on par with eight-stack HBM2. Overall, increasing the write latency to 1 μs reduces throughput by 17%, whereas increasing it to 2 μs reduces throughput by 39% on average.

5.6.2 Co-Design Area Benefit. All of our results so far have assumed the co-designed L2-ReRAM main memory module. As mentioned in Section 5.1, without co-design, each tile would be 22.1%

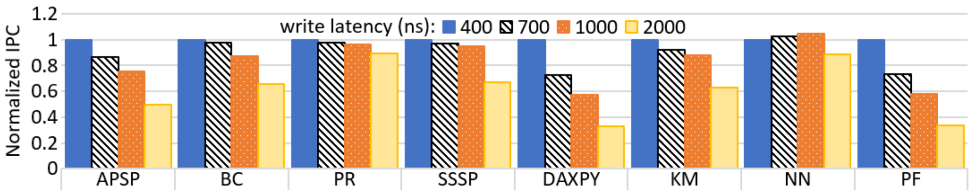

Fig. 18. Performance affect when increasing the write latency.

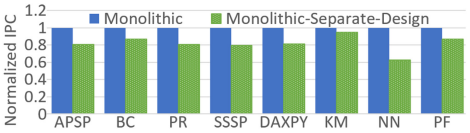

Fig. 19. Co-designed versus a separately designed system (approximated).

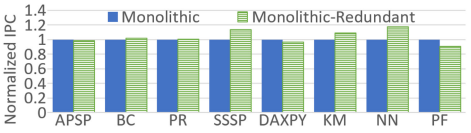

Fig. 20. Streamlined LLC/main memory interface versus redundant interconnects and controllers.

larger: instead of 256 cores, we would only have 210 cores in the same die area. Unfortunately, our simulator requires a power-of-2 number of tiles, so we cannot evaluate 210 cores. To estimate the impact, we reduced each core's threads from 4 to 3 (25% reduction), and we reduced the number of per-tile ReRAM banks from 128 to 112 (14% reduction) to maintain a divisible-by-8 match with cache lines. Figure 19 shows the performance of this reduced configuration, labeled "Monolithic-Separate-Design," normalized to the 1,024-thread "Monolithic" bars from Figure 12. Figure 19 shows that on average, performance decreases by 18% for the reduced configuration. Although this is not a precise experiment, it nevertheless shows that our co-designed L2-main memory structure affords area efficiency that can directly translate into performance gains.

5.6.3 Shared Interconnect and Unified Controller Benefit. All of our results so far have assumed the streamlined LLC/main memory interface. We ran a set of experiments that do not use this streamlined interface—that is, assuming separate internal interconnects for the LLC and ReRAM main memory, and assuming a separate cache controller and MC. Figure 20 shows the performance of this memory system with redundant interfaces and separate controllers, labeled "Monolithic-Redundant," normalized to the 1,024-thread "Monolithic" bars from Figure 12. The streamlined and redundant systems exhibit very similar performance, with an average of 4% better performance for the redundant case. As mentioned earlier, the streamlined interface may increase contention due to the shared internal interconnect, which is why performance is better for the redundant system in BC, PR, SSSP, KM, and NN. At the same time, the streamlined interface has lower latency, which is why performance degrades slightly for the redundant system in DAXPY and PF.

5.6.4 Variable Granularity Benefit. Our multi-grain fetch support benefits applications with sparse memory access patterns, namely the graph kernels and NN. To quantify this benefit, we re-ran our monolithic system experiments without this support—that is, *all* cache misses request a full cache block rather than scatter-gather misses requesting only 8 bytes. We do not change the underlying architecture, meaning that requests incur a small seven-cycle overhead to sequentially send the request to each of the eight ReRAM banks. Figure 21 shows the performance of this coarse-grained memory system, labeled "Monolithic-Coarse-Grain," normalized to the 1,024-thread "Monolithic" bars from Figure 12. Averaged across the five kernels with sparse memory access patterns, the throughput is reduced by 20% without multi-grain support. (Although not shown, the multi-grain support also improves dynamic energy consumption by 32%.) Compared

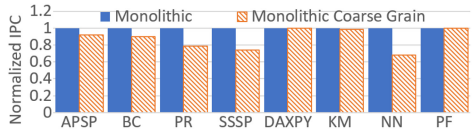

Fig. 21. Variable fetch granularity versus always fetching 64-byte cache blocks.

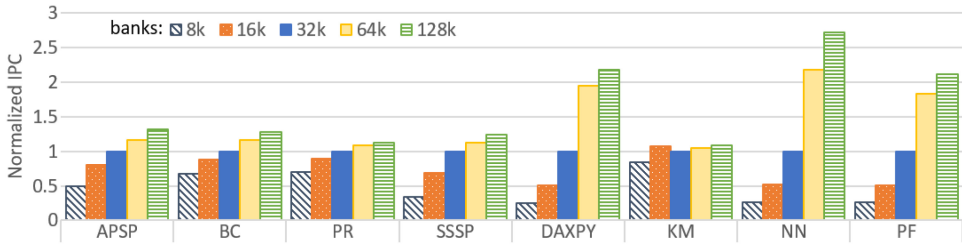

Fig. 22. Performance effect when scaling the number of ReRAM banks.

to always accessing 64-byte cache blocks, the finer-grained 8-byte accesses provided by our multi-grain support afford greater parallelism and eliminate wastefully fetched data for sparse memory access patterns.

5.6.5 Technology Scaling Impact. Our selected design point includes 32,000 ReRAM banks in the memory system, which is our estimate for the 14-nm technology node with 2-MB L2 cache slices. This is a lot of memory parallelism. However, ReRAM is also highly scalable. In future technology nodes (e.g., 7 nm), it is conceivable to have up to 128,000 banks. The given design point also has a large amount of cache—512 MB. If the area dedicated to cache and the memory system were reduced in favor of computational logic, the number of ReRAM banks would also be reduced. An important question is how might scaling the number of banks impact our performance results?

Figure 22 reports the throughput on a monolithic computer when the number of ReRAM banks is swept from 8,000 to 128,000 banks. For each benchmark, the performance is normalizaed to the baseline configuration of 32,000 banks. For all of the experiments, we run the benchmarks with 1,024 threads and employ a read latency of 200 ns (and a write latency of 400 ns).

As Figure 22 shows, scaling up the number of ReRAM banks has a positive impact on performance for all benchmarks but KM, which is compute bound. The reason for the improvement, however, depends on the type of benchmark. For the graph kernels, performance improves primarily because of a reduction in bank conflicts. We find that the scatter-gather requests in the graph kernels are uncorrelated, so they tend to spread across all of the banks. Even though the tiled CPU that we assume is not capable of completely saturating all 32,000 ReRAM banks, the sheer volume of memory requests means there still exists a significant probability for accesses to collide. The left half of Figure 22 confirms this: when the number of banks is increased from 32,000 to 128,000, the graph kernels' performance improves by 24.4% on average.

In contrast, for the streaming computations, performance improves primarily because of an increase in total memory bandwidth. As described earlier, the peak read bandwidth for the baseline configuration is 1.2 TB/s (and is lower with writes factored in). When the number of banks increases to 128,000, this peak read bandwidth quadruples to 4.8 TB/s. The streaming computations running on our tiled CPU cannot utilize this amount of memory bandwidth, so we do not expect performance to quadruple, but before that point the performance scaled fairly linearly with banks.

As the right half of Figure 22 shows, the streaming computations exhibit a 102% improvement in performance on the scaled-up configuration.

5.6.6 Cache Coherence Impact. Many-core CPUs with a two-level cache hierarchy and a shared L2 usually maintain coherence on the private L1s. Although our design in Sections 3 and 4 considered directories, we did not simulate cache coherence in our evaluation, opting for non-coherent L1s more akin to accelerators. However, we did analyze potential impact. In our simulator, we counted the number of L2 accesses that do not require coherence: read hit with no writer, write hit with no sharers, and miss to ReRAM. Averaged over all kernels, these counts make up 99.3% of all L2 accesses. For the worst-case benchmark, the counts make up 96.1% of all L2 accesses. The memory-intensive nature of our kernels means cache blocks are short lived in the CPU caches, so there is very little dynamic sharing. From this analysis, we conclude that our results would not change appreciably for the design with directories. The main impact of cache coherence would be eliminating the need to manually enforce coherence.

6 RELATED WORK

Gong [17] has designed a SRAM/MRAM hybrid TLB-Cache that makes use of monolithic 3D integration of non-voltatile memories. This work looks to increase the capacity and decrease the power consumption of the cache by leveraging similar techniques to our main memory system.

Sabry Aly's group [3, 4] proposed monolithically integrating STT-RAM and ReRAM with carbon nanotubes and CMOS transistors. However, their approach relies on advanced 3D processing technology that can fabricate both compute and memory devices across multiple vertical layers of silicon. In contrast, our work proposes stacking ReRAM in BEOL metal layers over CPU logic that is implemented using planar transistors. This can be fabricated in today's commercial CMOS technology and is a more practical approach for integrating main memory with the CPU chip.

Our own previous work, Jagasivamani et al. [23–25], was the first, to the best of our knowledge, to propose integrating ReRAM over cache, but that early version only showed that individual cache mats can fit underneath ReRAM subarrays. This article provides the design of a complete integrated ReRAM memory module and cache slice, including a coherence directory, and optimizes the integrated 3D structure for area, delay, and power. The current work is also the first, to the best of our knowledge, to propose a unified cache/memory controller and shared internal interconnects. In addition, it incorporates our 3D LLC slice/main memory module into a variable granularity memory system for a many-core CPU and fully evaluates the entire integrated architecture.

Besides monolithically integrated ReRAM, there are other integration methods that researchers have explored. Embedded DRAM [28] can also be integrated into the CPU die, although at far lower densities than what we propose with ReRAM. Another approach is stacking memory dies directly on top of the CPU [42, 50]. However, as discussed in Section 1, die stacks provide a limited number of TSVs since the TSVs are considerably larger than the vias that can be used with monolithic integration. The TSVs are generally placed centrally on the chip to prevent disrupting the layout of other logic, as shown in Figure 1. This increases data movement and energy, and reduces the overall bandwidth and performance of the system. Moreover, to achieve comparable bandwidths, the switching speeds need to be much higher over the smaller number of TSV connections, further increasing power consumption. Finally, Xu et al. [47] also studied a ReRAM-based main memory system, but it was discrete and not monolithically integrated with the CPU.

7 CONCLUSION

This article studies monolithically integrating ReRAM main memory over the CPU's LLC. We exploit ReRAM's 3D crosspoint architecture to recoup unused area underneath the subarrays for

implementing the LLC. Using Cacti, we co-designed the LLC's mats, subarrays, and banks with ReRAM's subarrays and banks to boost area efficiency. Our design saves 27% of the total LLC and main memory area. We also integrated a sparse full-map directory alongside each L2 slice underneath the ReRAM that is large enough for a 32- to 128-KB L1 cache. In addition, we developed a streamlined LLC/main memory interface that employs a shared internal interconnect and a unified controller to service both LLC and main memory requests. These optimizations save an additional 12% in area. Our results show that monolithic integration of CPU and main memory improves performance by 5.3× and 1.7× over HBM2 DRAM for several graph and streaming kernels, respectively. It also reduces the memory system's energy by 6.0× and 1.7×, respectively. We also show that the area savings of co-design allows another 23% in cores and main memory, and that streamlining the LLC/main memory interface incurs a small 4% performance penalty.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*.

[2] Masab Ahmad, Farrukh Jijaz, Qingchuan Shi, and Omer Khan. 2015. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*.

[3] Mohamed M. Sabry Aly, Mingyu Gao, Gage Hills, Chi-Shuen Lee, Greg Pitner, Max M. Shulaker, Tony F. Wu, et al. 2015. Energy-efficient abundant-data computing: The N3XT 1, 000×. *Computer* 48, 12 (Dec. 2015), 24–33.

[4] Mohamed M. Sabry Aly, Tony F. Wu, Andrew Bartolo, Yash H. Malviya, William Hwang, Gage Hills, Igor Markov, et al. 2019. The N3XT approach to energy-efficient abundant-data computing. *Proceedings of the IEEE* 107, 1 (Jan. 2019), 19–48.

[5] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, and Theresa Meuse. 2006. HPCS Scalable Synthetic Compact Applications #2 Graph Analysis. Retrieved May 28, 2021 from http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.1.pdf.

[6] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'11)*. 1–12. https://doi.org/10.1145/2063384.2063454

[7] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization* 11, 3 (April 2014), Article 28.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*.

[10] Leon O. Chua. 1971. Memristor—The missing circuit element. *IEEE Transactions on Circuit Theory* 18, 5 (1971), 507–519. https://doi.org/doi:10.1109/TCT.1971.1083337

[11] Crossbar. 2017. ReRAM Memory, Crossbar. https://www.crossbar-inc.com/assets/resources/white-papers/Crossbar-ReRAM-Technology.pdf.

[12] Crossbar. 2020. Personal communication.

[13] Ian Cutress. 2015. SuperComputing 15: Intel's Knights Landing/Xeon Phi Silicon on Display. Retrieved May 28, 2021 from https://www.anandtech.com/show/9802/supercomputing-15-intels-knights-landing-xeon-phi-silicon-on-display.

[14] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the Design Automation Conference*.

[15] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. 2012. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (July 2012), 994–1007.

[16] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems*.

[17] Young-Ho Gong. 2021. Monolithic 3D-based SRAM/MRAM hybrid memory for an energy-efficient unified L2 TLB-cache architecture. *IEEE Access* 9 (2021), 18915–18926. https://doi.org/10.1109/ACCESS.2021.3054021

[18] Anoop Gupta, Wolf Dietrich Weber, and Todd Mowry. 1990. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the International Conference on Parallel Processing*. 312–321.

[19] Charlie Demerjian. 2004. Sun's Niagara falls neatly into multithreaded place. The Inquirer, 02 November 2004.

[20] Intel. 2012. Intel Software Development Emulator. Retrieved May 28, 2021 from http://software.intel.com/en-us/articles/intel-software-development-emulator.

[21] Intel. 2017. AVX 512 Instruction Extensions. Retrieved May 28, 2021 from http://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[22] Intel. 2017. Intel Optane Technology. Retrieved May 28, 2021 from http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[23] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. 2020. Tileable monolithic ReRAM memory design. In *Proceedings of the IEEE Symposium on Low-Power and High-Speed Chips and Systems*.

[24] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Bruce Jacob, and Donald Yeung. 2019. Analyzing the monolithic integration of a ReRAM-based main memory into a CPU's die. *IEEE Micro* 39, 6 (Nov.-Dec. 2019), 64–72.

[25] Meenatchi Jagasivamani, Candace Walden, Devesh Singh, Luyi Kang, Shang Li, Mehdi Asnaashari, Sylvain Dubois, Donald Yeung, and Bruce Jacob. 2019. Design for ReRAM-based main-memory architectures. In *Proceedings of the 5th International Symposium on Memory Systems*.

[26] Sung Hyun Jo, Kuk-Hwan Kim, and Wei Lu. 2009. High-density cross-bar arrays based on a Si memristive system. *Nano Letters* 9, 2 (2009), 870–874.

[27] Sung Hyun Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian. 2014. 3D-stackable crossbar resistive memory based on field assisted superlinear threshold (FAST) selector. In *Proceedings of the IEEE International Electron Devices Meeting*.

[28] Doris Keitel-Schulz and Norbert Wehn. 2001. Embedded DRAM development: Technology, physical design, and application issues. *IEEE Design & Test of Computers* 18, 3 (May-June 2001), 7–15.

[29] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. 2009. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the International Symposium on Computer Architecture*. 140–151.

[30] Myoung-Jae Lee, Chang Bum Lee, Dongsoo Lee, Seung Ryul Lee, Man Chang, Ji Hyun Hur, Young-Bae Kim, et al. 2011. A fast, high-endurance and scalable non-volatile memory device made from asymmetric Ta2O5-x/TaO2-x bilayer structures. *Nature Materials* 10 (Aug. 2011), 625–630.

[31] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. 2014. CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Transactions on Computers* 63, 9 (Sept. 2014), 2187–2200.

[32] Sukhan Lee, HyunYoon Cho, Young Hoon Son, Yuhwan Ro, Nam Sung Kim, and Jung Ho Ahn. 2018. Leveraging power-performance relationship of energy-efficient modern DRAM devices. *IEEE Access* 6 (June 2018), 31387–31398.

[33] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*.

[34] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2019. DRAMsim3: A cycle-accurate, thermal capable memory system simulator. *IEEE Computer Architecture Letters* 19, 2 (2019), 106–109.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[36] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*.

[37] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. IEEE, Los Alamitos, CA, 3–14. https://doi.org/10.1109/MICRO.2007.30

[38] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: Energy-efficient DRAM for extreme bandwidth systems. In *Proceedings of the 50th International Symposium on Microarchitecture*.

[39] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual International Symposium on Microarchitecture*.

[40] Moinuddin K. Qureshi, Vijayalakshmi, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture*.
[41] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the 2011 International Conference on Supercomputing*.
[42] Parthasarathy Ranganathan. 2011. From microprocessors to nanostores: Rethinking data-centric systems. *Computer* 44, 1 (Jan. 2011), 39–48.
[43] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th International Symposium on Computer Architecture*.
[44] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, et al. 2008. Larrabee: A many-core ×86 architecture for visual computing. *ACM Transactions on Graphics* 27, 3 (Aug. 2008), 1–16.
[45] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman Jouppi. 2008. *CACTI 5.1*. Technical Report. HP Laboratories.
[46] Dmitrii Ustiugov, Alexandros Daglis, Javier Picorel, Mark Sutherland, Edouard Bugnion, Babak Falsafi, and Dionisios Pnevmatikatos. 2018. Design guidelines for high-performance SCM hierarchies. In *Proceedings of the 4th International Symposium on Memory Systems*.
[47] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*.
[48] D. H. Yoon, M. K. Jeong, and M. Erez. 2011. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *Proceedings of the 2011 38th Annual International Symposium on Computer Architecture (ISCA'11)*. 295–306.
[49] Lunkay Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. 2016. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*.
[50] Wangyuan Zhang and Tao Li. 2009. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the International Symposium on Parallel Architectures and Compilation Techniques*.

Received October 2020; revised April 2021; accepted April 2021